TURING 图灵程序设计丛书

[PACKT] PUBLISHING

[乌拉圭] Fernando Doglio 著 陶俊杰 陈小莉 译

Python 性能分析与优化

Mastering Python High Performance

全面掌握Python代码性能分析和优化方法，消除性能瓶颈，迅速改善程序性能！



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Fernando Doglio

Globant公司软件架构师。过去十年一直从事Web开发工作，期间使用了大多数最前沿的技术，如PHP、Ruby on Rails、MySQL、Python、Node.js、AngularJS、REST API等。Fernando喜欢钻研新事物，他的GitHub账户每个月也会因此获得回购。他还是开源拥护者，并通过网站lookingforpullrequests.com来获得人们的支持。Fernando另著有*Pro REST API Development with Node.js*。他的Twitter账号是@deleteman123。

陶俊杰

长期从事数据分析工作，酷爱Python，每天都和Python面对面，乐此不疲。本科毕业于北京交通大学机电学院，硕士毕业于北京交通大学经管学院。曾就职于中国移动设计院，目前在京东任职。

陈小莉

长期从事数据分析工作，喜欢Python。本科与硕士毕业于北京交通大学电信学院。目前在中科院从事科技文献与专利分析工作。



[乌拉圭] Fernando Doglio 著 陶俊杰 陈小莉 译

Python 性能分析与优化

Mastering Python High Performance

人民邮电出版社
北 京

图书在版编目（CIP）数据

Python性能分析与优化 / （乌拉圭）费尔南多·多格里奥（Fernando Doglio）著 ; 陶俊杰，陈小莉译. -- 北京 : 人民邮电出版社，2016.6
（图灵程序设计丛书）
ISBN 978-7-115-42422-8

Ⅰ. ①P… Ⅱ. ①费… ②陶… ③陈… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2016)第108957号

内 容 提 要

本书首先介绍什么是性能分析，性能分析如何在项目开发周期中发挥作用，以及通过在项目中进行性能分析实践能够取得的效果。紧接着介绍分析性能所需的核心工具（性能分析器和可视化性能分析器）。然后介绍一系列性能优化技术，最后一章会介绍一个具有实际意义的优化案例。

本书适合所有 Python 程序员阅读。

◆ 著 [乌拉圭] Fernando Doglio
译 陶俊杰 陈小莉
责任编辑 岳新欣
执行编辑 李 敏
责任印制 彭志环

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：12
字数：262千字 2016年6月第1版
印数：1－3 500册 2016年6月北京第1次印刷

著作权合同登记号 图字：01-2016-2262号

定价：45.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

译　者　序

从狭义相对论的角度看，速度最快、规模最大的并行计算方式是太阳照耀地球。每时每刻，阳光都会离开太阳表面，以大约30万千米/秒的速度，经过8分17秒到达地球表面。太阳的计算方式很简单，一视同仁，普照大地，并行（parallel）照耀每一个对象，谁也不会多得一米阳光。地球上的每个人都可看成享受阳光资源的独立进程（process），人们平时处理自己的任务，经历着各自的生命周期，彼此间有时也会通信（进程间通信，IPC）。由于太阳资源丰富，可以不计得失，她也许从来不觉得自己的光具有波粒二象性，也没觉得并行计算的效率高。

但是，人不是太阳，每个人在一生中时刻面对着诸多问题。“人无远虑，必有近忧”，本质上皆为时间与空间的稀缺问题，这与计算机的多任务处理问题一致。完成任务之前，需要精打细算，以期充分利用资源，尽可能地多快好省，实现高效运行。对于单任务，人们会努力提升自己的能力，并借助高性能的工具，提高做事的效率，从内部不断优化自己。对于多个简单任务，可以完成一个任务再去完成另一个任务，就像`for`循环处理方式。然而，“好汉难敌四手”，个人的力量总是有限的，所以时间紧迫、任务艰巨时，团队的力量（多进程）不可或缺。有时，在处理没有共同资源需求的多个任务时，可以多人同时作业，并行处理，提高效率。处理有共同资源需求的多个任务时，我们会为每个人设定不同的阶段性目标，在这段时间做点任务A，在那段时间做点任务B，也就是多线程的并发（concurrency）处理；这样，一段时间后，两个任务就都可以完成。然而，多进程与多线程，究竟哪种方式效率高、效果好，需要根据实际情况决定。

计算机多任务处理的理论，与上述情景类似。大师们已经将这些资源配置方法抽象成完整的理论，不仅适用于计算机编程语言，对提高个人素质和改善团队合作方式也大有裨益。众所周知，Python语言简洁优雅的特点，使其生产效率大幅提升，然而在面对海量的数据处理、文本分析、服务器响应时，其性能瓶颈也十分明显。但是，Python社区一直在努力，从语言自身的特性到计算机优化理论，特别是多任务处理（并行、并发、分布式）等方面，不断地改善Python的性能。本书即是社区对Python性能分析与优化实践的系统性总结之一。

本书内容丰富，浅显易懂，适合有Python基础的读者阅读。作者从算法性能分析理论开始，首先介绍主流的Python性能分析工具，包括`cProfile`性能分析器、`line_profiler+kernprof`性能分析工具、KCacheGrind+`pyprof2calltree`、RunSnakeRun可视化性能分析工具，帮助读者发现程序的性能瓶颈。紧接着，将通用性能优化方法与Python语言结构紧密结合起来，优化程

序的性能，介绍了函数值缓存、列表生成器、ctypes和字符串优化等技巧。之后，介绍了Python多线程与多进程的多任务处理方法，并对PyPy（JIT编译器）与Cython（引入C语言类型）的用法与特点进行了深入分析。另外，针对Python在数据分析领域的重要地位，作者还专门介绍了高性能的数据处理程序库，如Numba、Parakeet和pandas。最后，作者通过一个Python网络爬虫案例，将前面介绍的性能分析与优化方法结合起来，不断地改善程序的性能，对比性能优化的效果。

优化Python也是需要成本的。写Python总是很happy，因为在普通的业务场景中，Python并不慢，再结合成熟的科学计算生态环境，问题大都可以轻松解决。虽然Python的性能分析与优化方法都很简单，但是优化也是需要花费时间的，所以大规模的性能优化需要面对特定的业务场景才有意义，正如不是所有人每天都需要跑100米冲刺，亦如Python的发明者Guido van Rossum所说，"……对绝大多数事情而言，语言性能并不重要……"（...for most of what you're doing, the speed of the language is irrelevant...）。假如你的Python程序总是需要优化，那么高性能部分代码可以考虑使用新语言实现，参考Google与Dropbox的发展轨迹。如果你有时间、想优化，那就动手吧，本书可以给你一些指导。不过，开源项目的发展速度不是书本可以跟上的，所以要想了解最新的进展，需要主动关注相关工具的官方网站或GitHub。书籍能够提供的是基础性、系统化的指导，知识可能过时，但方向不会改变。

Python生态系统的性能与时俱进。随着数据获取成本与云计算资源成本的不断降低，人们越来越容易获取强大的计算资源（如亚马逊AWS、微软Azure、谷歌云计算平台）。PEP不断地引入优质方案，增强Python的性能，Spark（支持Python）、dask、IPython的`ipyparallel`等并行计算框架也在快速发展，Python的生态系统性能不断地优化，精彩的故事正在进行，让我们共同努力吧（Let's do more of this）。

简单，所以持久。

陶俊杰

于2016年5月

前　　言

Packt出版社的朋友们让我产生了写作本书的想法。他们希望有人可以深入探讨Python高性能这个错综复杂的问题，介绍与之相关的所有话题，包括代码性能分析、现有的性能分析工具（比如性能分析器和其他性能增强技术），甚至包括标准Python实现的其他版本。

因此，欢迎你阅读本书。在本书中，我们将介绍与程序性能优化有关的一切。与性能优化这个主题相关的知识并不是阅读本书的必要前提（当然了解也没有坏处），但是关于Python编程语言的知识是必需的，尤其是对于一些专门优化Python代码的章节而言。

我们首先将会介绍什么是性能分析，性能分析如何在项目开发周期中发挥作用，以及通过在项目中进行性能分析实践能够取得的效果。紧接着将介绍分析性能所需的核心工具（性能分析器和可视化性能分析器）。然后会讨论一系列性能优化技术，最后一章会介绍一个具有实际意义的优化示例。

本书内容

第1章，性能分析基础，为没有关注过性能分析艺术的人们介绍相关的基础知识。

第2章，性能分析器，介绍如何使用贯穿全书的核心分析工具。

第3章，可视化——利用GUI理解性能分析数据，介绍如何使用KCacheGrind/`pyprof2calltree`和RunSnakeRun工具，并且通过不同的可视化技术帮助开发者理解`cProfile`的输出结果。

第4章，优化每一个细节，介绍性能优化的基本过程和一系列值得推荐的好习惯，每个Python开发者在尝试其他优化手段之前都应该优先采用这些做法。

第5章，多线程与多进程，介绍多线程和多进程，并论述二者的使用方法和适用场景。

第6章，常用的优化方法，介绍如何安装和使用Cython和PyPy，优化代码性能。

第7章，用Numba、Parakeet和pandas实现极速数据处理，介绍针对处理数据的Python脚本的性能优化工具。这些专用工具（Numba、Parakeet和pandas）可以提升数据处理效率。

第8章，付诸实践，提供一个性能分析的实际示例，探索性能瓶颈，并通过书中介绍过的工具和技术消除瓶颈。总之，我们将会利用每项技术对比结果。

本书需要的工具

在运行本书中的代码之前，你的操作系统中必须安装以下工具：

- Python 2.7
- line_profiler 1.0b2
- KCacheGrind 0.7.4
- RunSnakeRun 2.0.4
- Numba 0.17
- 最新版本的Parakeet
- pandas 0.15.2

目标读者

由于本书涵盖了Python代码性能分析和优化的方方面面，所以不同水平的Python开发者都能从中受益。

唯一的要求是读者要具备Python编程语言的一些基础知识。

排版约定

本书中用不同的文本样式来区分不同种类的信息。下面给出了这些文本样式的示例及其含义。

正文中的代码和用户输入会这样显示："我们可以打印/收集`PROFILER`函数里我们觉得有意义的内容。"

代码块示例如下：

```
import sys

def profiler(frame, event, arg):
    print 'PROFILER: %r %r' % (event, arg)

sys.setprofile(profiler)
```

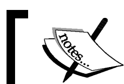
当我们希望你注意代码块中的某些部分时，相关的行或者文字会被加粗：

```
Traceback (most recent call last):
    File "cprof-test1.py", line 7, in <module>
        runRe() ...
    File "/usr/lib/python2.7/cProfile.py", line 140, in runctx
        exec cmd in globals, locals
    File "<string>", line 1, in <module>
NameError: name 're' is not defined
```

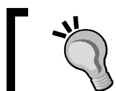
命令行输入或输出将会这样表示：

```
$ sudo apt-get install python-dev libxml2-dev libxslt-dev
```

新术语和重点词汇均采用楷体字表示。



这个图标表示警告或需要特别注意的内容。



这个图标表示提示或者技巧。

读者反馈

我们非常欢迎读者的反馈。如果你对本书有些想法，有什么喜欢或是不喜欢的，请反馈给我们。这将有助于我们开发出能够充分满足读者需求的图书。

一般的反馈，请发送电子邮件至feedback@packtpub.com，并在邮件主题中注明书名。

如果你在某个领域有专长，并有意编写一本书或是贡献一份力量，请参考我们的作者指南，地址为http://www.packtpub.com/authors。

客户支持

你现在已经是Packt引以为傲的读者了，为了能让你的购买物有所值，我们还为你准备了以下内容。

下载示例代码

你可以用你的账户从http://www.packtpub.com下载所有已购买Packt图书的示例代码文件。如果你从其他地方购买本书，可以访问http://www.packtpub.com/support并注册，我们将通过电子邮件把文件发送给你。

下载本书的彩色图像

我们也提供了本书的PDF文件，里面包含了本书的截屏和流程图等彩色图片。彩色图片将能帮助你更好地理解输出的变化。你可以通过https://www.packtpub.com/sites/default/files/downloads/9300OS_GraphicBundle.pdf下载。

勘误

虽然我们已尽力确保本书内容正确，但出错仍旧在所难免。如果你在我们的书中发现错误，不管是文本还是代码，希望能告知我们，我们不胜感激。这样做，你可以使其他读者免受挫败，帮助我们改进本书的后续版本。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata提交，选择你的书，点击勘误表提交表单的链接，并输入详细说明。勘误一经核实，你的提交将被接受，此勘误将上传到本公司网站或添加到现有勘误表。从http://www.packtpub.com/support选择书名就可以查看现有的勘误表。

侵权行为

版权材料在互联网上的盗版是所有媒体都要面对的问题。Packt非常重视保护版权和许可证。如果你发现我们的作品在互联网上被非法复制，不管以什么形式，都请立即为我们提供位置地址或网站名称，以便我们可以寻求补救。

请把可疑盗版材料的链接发到copyright@packtpub.com。

非常感谢你帮助我们保护作者，以及保护我们给你带来有价值内容的能力。

问题

如果你对本书内容存有疑问，不管是哪个方面，都可以通过questions@packtpub.com联系我们，我们将尽最大努力来解决。

电子书

扫描如下二维码，即可获得本书电子版。



致　　谢

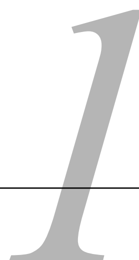
我想感谢我挚爱的妻子忍受我花这么长时间来写这本书，如果没有她的支持，我不可能完成这本书。我也想感谢我的两个儿子，没有他们，这本书就不会提前数月完成。

最后，我想感谢所有审稿人和编辑们。他们帮助我使本书成形，并且达到了较高的质量水平。

目　　录

第 1 章

性能分析基础



就像在12秒内跑完100米障碍跑的人在婴儿时期需要先学爬一样，程序员在精通性能分析（profiling）之前需要先了解一些基础知识。因此，在我们探索Python程序的性能优化与分析技术之前，需要对相关的基础知识有一个清晰的认识。

只要你掌握了这些基础知识，就可以进一步学习具体的工具和技术。因此，这一章将介绍所有你平时羞于开口问人却又应该掌握的性能分析知识。本章的具体内容如下。

- 介绍性能分析的明确定义，概述各种性能分析技术。
- 论述性能分析在开发周期中的重要作用，因为性能分析不是那种只做一次就抛到脑后的事情。性能分析应该是开发过程中一个完整的组成部分，就像写测试一样。
- 介绍哪些东西适合进行性能分析。看看我们可以度量哪些资源，以及这些度量如何帮助我们发现性能瓶颈。
- 分析过早优化的风险，即解释为什么未经性能分析便对代码进行优化通常不是一种好做法。
- 学习关于程序运行时间复杂性的知识。虽然理解性能分析技术是成功优化程序的一个步骤，但我们也需要理解算法复杂性的度量指标，这样才能够明白是否有必要优化算法。
- 一些好的做法。本章最后将介绍一些对项目进行性能分析时需要记住的好习惯。

1.1 什么是性能分析

没有优化过的程序通常会在某些子程序（subroutine）上消耗大部分的CPU指令周期（CPU cycle）。性能分析就是分析代码和它正在使用的资源之间有着怎样的关系。例如，性能分析可以告诉你一个指令占用了多少CPU时间，或者整个程序消耗了多少内存。性能分析是通过使用一种被称为性能分析器（profiler）的工具，对程序或者二进制可执行文件（如果可以拿到）的源代码进行调整来完成的。

通常，当需要优化程序性能，或者程序遇到了一些奇怪的bug时（一般与内存泄漏有关），开发者会对他们的程序进行性能分析。这时，性能分析可以帮助开发者深刻地了解程序是如何使用

计算机资源的（即可以细致到一个函数被调用了多少次）。

根据这些信息，以及对源代码的深刻认知，开发者就可以找到程序的性能瓶颈或者内存泄漏所在，然后修复错误的代码。

性能分析软件有两类方法论：基于事件的性能分析（event-based profiling）和统计式性能分析（statistical profiling）。在使用这两类软件时，应该牢记它们各自的优缺点。

1.1.1 基于事件的性能分析

不是所有的编程语言都支持这类性能分析。支持这类基于事件的性能分析的编程语言主要有以下几种。

- Java：JVMTI（JVM Tools Interface，JVM工具接口）为性能分析器提供了钩子，可以跟踪诸如函数调用、线程相关的事件、类加载之类的事件。
- .NET：和Java一样，.NET运行时提供了事件跟踪功能（https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing/Profiling#Methods_of_data_gathering）。
- Python：开发者可以用`sys.setprofile`函数，跟踪`python_[call|return|exception]`或`c_[call|return|exception]`之类的事件。

基于事件的性能分析器（event-based profiler，也称为轨迹性能分析器，tracing profiler）是通过收集程序执行过程中的具体事件进行工作的。这些性能分析器会产生大量的数据。基本上，它们需要监听的事件越多，产生的数据量就越大。这导致它们不太实用，在开始对程序进行性能分析时也不是首选。但是，当其他性能分析方法不够用或者不够精确时，它们可以作为最后的选择。如果你想分析程序中所有返回语句的性能，那么这类性能分析器就可以为你提供完成任务应该有的颗粒度，而其他性能分析器都不能为你提供如此细致的结果。

一个Python基于事件的性能分析器的简单示例代码如下所示（当学完后面的章节时，你对这个主题的理解将会更加深刻）：

```
import profile
import sys

def profiler(frame, event, arg):
    print 'PROFILER: %r %r' % (event, arg)

sys.setprofile(profiler)

# 计算斐波那契数列的简单（也是非常低效的）示例
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
```

```
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = [ ]
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

print fib_seq(2)
```

上面程序的输出结果如下所示：

```
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'return' 0
PROFILER: 'c_call' <built-in method append of list object at 0x7f570ca215f0>
PROFILER: 'c_return' <built-in method append of list object at 0x7f570ca215f0>
PROFILER: 'return' [0]
PROFILER: 'c_call' <built-in method extend of list object at 0x7f570ca21bd8>
PROFILER: 'c_return' <built-in method extend of list object at 0x7f570ca21bd8>
PROFILER: 'call' None
PROFILER: 'return' 1
PROFILER: 'c_call' <built-in method append of list object at 0x7f570ca21bd8>
PROFILER: 'c_return' <built-in method append of list object at 0x7f570ca21bd8>
PROFILER: 'return' [0, 1]
PROFILER: 'c_call' <built-in method extend of list object at 0x7f570ca55bd8>
PROFILER: 'c_return' <built-in method extend of list object at 0x7f570ca55bd8>
PROFILER: 'call' None
PROFILER: 'call' None
PROFILER: 'return' 1
PROFILER: 'call' None
PROFILER: 'return' 0
PROFILER: 'return' 1
PROFILER: 'c_call' <built-in method append of list object at 0x7f570ca55bd8>
PROFILER: 'c_return' <built-in method append of list object at 0x7f570ca55bd8>
PROFILER: 'return' [0, 1, 1]
[0, 1, 1]
PROFILER: 'return' None
PROFILER: 'call' None
PROFILER: 'c_call' <built-in method discard of set object at 0x7f570ca8a960>
PROFILER: 'c_return' <built-in method discard of set object at 0x7f570ca8a960>
PROFILER: 'return' None
PROFILER: 'call' None
PROFILER: 'c_call' <built-in method discard of set object at 0x7f570ca8f3f0>
PROFILER: 'c_return' <built-in method discard of set object at 0x7f570ca8f3f0>
PROFILER: 'return' None
```

你会发现，`PROFILER`会被每一个事件调用。我们可以打印/收集`PROFILER`函数里我们觉得有意义的内容。在上面的简单示例代码中，最后一行表示执行`fib_seq(2)`生成一组数值。如果

我们处理一个实际点儿的程序，性能分析输出的结果可能要比上述结果大好几个数量级。这就是基于事件的性能分析软件通常作为性能分析的最后选择的原因。虽然其他性能分析软件（马上就会看到）产生的结果会少很多，但是分析的精确程度也要低一些。

1.1.2 统计式性能分析

统计式性能分析器以固定的时间间隔对程序计数器（program counter）进行抽样统计。这样做可以让开发者掌握目标程序在每个函数上消耗的时间。由于它对程序计数器进行抽样，所以数据结果是对真实值的统计近似。不过，这类软件足以窥见被分析程序的性能细节，查出性能瓶颈之所在。

这类性能分析软件的优点如下所示。

- **分析的数据更少**：由于我们只对程序执行过程进行抽样，而不用保留每一条数据，因此需要分析的信息量会显著减少。
- **对性能造成的影响更小**：由于使用抽样的方式（用操作系统中断），目标程序的性能遭受的干扰更小。虽然使用性能分析器并不能做到100%无干扰，但是统计式性能分析器比基于事件的性能分析器造成的干扰要小。

下面是一个Linux统计式性能分析器OProfile（http://oprofile.sourceforge.net/news/）的分析结果：

```
Function name,File name,Times Encountered,Percentage
"func80000","statistical_profiling.c",30760,48.96%
"func40000","statistical_profiling.c",17515,27.88%
"func20000","static_functions.c",7141,11.37%
"func10000","static_functions.c",3572,5.69%
"func5000","static_functions.c",1787,2.84%
"func2000","static_functions.c",768,1.22%
"func1500","statistical_profiling.c",701,1.12%
"func1000","static_functions.c",385,0.61%
"func500","statistical_profiling.c",194,0.31%
```

下面的性能分析结果，是通过Python的统计式性能分析器statprof对前面的代码进行分析得出的：

```
%     cumulative      self
time      seconds    seconds  name
100.00       0.01       0.01  B02088_01_03.py:11:fib
  0.00       0.01       0.00  B02088_01_03.py:17:fib_seq
  0.00       0.01       0.00  B02088_01_03.py:21:<module>
---
Sample count: 1
Total time: 0.010000 seconds
```

你会发现，两个性能分析器对同样代码的分析结果差异非常大。

1.2　性能分析的重要性

现在我们已经知道了性能分析的涵义，还应该理解在产品开发周期中进行性能分析的重要性和实际意义。

性能分析并不是每个程序都要做的事情，尤其对于那些小软件来说，是没多大必要的（不像那些杀手级嵌入式软件或专门用于演示的性能分析程序）。性能分析需要花时间，而且只有在程序中发现了错误的时候才有用。但是，仍然可以在此之前进行性能分析，捕获潜在的bug，这样可以节省后期的程序调试时间。

在硬件变得越来越先进、越来越快速且越来越便宜的今天，开发者自然也越来越难以理解，为什么我们还要消耗资源（主要是时间）去对开发的产品进行性能分析。毕竟，我们已经拥有测试驱动开发、代码审查、结对编程，以及其他让代码更加可靠且符合预期的手段。难道不是吗？

然而，我们没有意识到的是，随着我们使用的编程语言越来越高级（几年间我们就从汇编语言进化到了JavaScript），我们愈加不关心CPU循环周期、内存配置、CPU寄存器等底层细节了。新一代程序员都通过高级语言学习编程技术，因为它们更容易理解而且开箱即用。但它们依然是对硬件和与硬件交互行为的抽象。随着这种趋势的增长，新的开发者越来越不会将性能分析作为软件开发中的一个步骤了。

让我们看看下面这种情景。

我们已经知道，性能分析是用来测量程序所使用的资源的。前面已经说过，资源正变得越来越便宜。因此，生产软件并让更多的客户使用我们的软件，其成本变得越来越低。

如今，随便开发一个软件就可以获得上千用户。如果通过社交网络一推广，用户可能马上就会呈指数级增长。一旦用户量激增，程序通常会崩溃，或者变得异常缓慢，最终被客户无情抛弃。

上面这种情况，显然可能是由于糟糕的软件设计和缺乏扩展性的架构造成的。毕竟，一台服务器有限的内存和CPU资源也可能会成为软件的瓶颈。但是，另一种可能的原因，也是被证明过许多次的原因，就是我们的程序没有做过压力测试。我们没有考虑过资源消耗情况；我们只保证了测试已经通过，而且乐此不疲。也就是说，我们目光短浅，结果就是项目崩溃夭折。

性能分析可以帮助我们避免项目崩溃夭折，因为它可以相当准确地为我们展示程序运行的情况，不论负载情况如何。因此，如果在负载非常低的情况下，通过性能分析发现软件在I/O操作上消耗了80%的时间，那么这就给了我们一个提示。有人可能觉得，在测试阶段程序运行很正常，在负载很重的情况下也应该不会有问题。想想内存泄漏的情况吧。在这种情况下，小测试是不会发现大负载里出现的bug的。但是，产品负载过重时，内存泄漏就会发生。性能分析可以在负载真的过重之前，为我们提供足够的证据来发现这类隐患。

1.3　性能分析可以分析什么

要想深入地理解性能分析，很重要的一点是明白性能分析方法究竟能够分析什么指标。因为测量是性能分析的核心，所以让我们仔细看看程序运行时可以测量的指标。

1.3.1　运行时间

做性能分析时，我们能够收集到的最基本的数值就是运行时间。整个进程或代码中某个片段的运行时间会暴露相应的性能。如果你对运行的程序有一些经验（比如说你是一个网络开发者，正在使用一个网络框架），可能很清楚运行时间是不是太长。例如，一个简单的网络服务器查询数据库、响应结果、反馈到客户端，一共需要100毫秒。但是，如果程序运行得很慢，做同样的事情需要花费60秒，你就得考虑做性能分析了。你还需要考虑不同场景的可比性。再考虑另一个进程：一个MapReduce任务把2TB数据存储到文件中要消耗20分钟，这时你可能不会认为进程很慢了，即使它比之前的网络服务器处理时间要长很多。

为了获得运行时间，你不需要拥有大量性能分析经验和一堆复杂的分析工具。你只需要把几行代码加入程序运行就可以了。

例如，下面的代码会计算斐波那契数列的前30位：

```
import datetime

tstart = None
tend = None

def start_time():
    global tstart
    tstart = datetime.datetime.now()

def get_delta():
    global tstart
    tend = datetime.datetime.now()
    return tend - tstart



def fib(n):
     return n if n == 0 or n == 1 else fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq
```

```
start_time()
print "About to calculate the fibonacci sequence for the number 30"
delta1 = get_delta()

start_time()
seq = fib_seq(30)
delta2 = get_delta()

print "Now we print the numbers: "
start_time()
for n in seq:
    print n
delta3 = get_delta()

print "====== Profiling results ======="
print "Time required to print a simple message: %(delta1)s" % locals()
print "Time required to calculate fibonacci: %(delta2)s" % locals()
print "Time required to iterate and print the numbers: %(delta3)s" % locals()
print "======  ======="
```

程序的输出结果如下所示：

```
About to calculate the Fibonacci sequence for the number 30
Now we print the numbers:
0
1
1
2
3
5
8
13
21
#……省略一些数字
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
====== Profiling results =======
Time required to print a simple message: 0:00:00.000030
Time required to calculate fibonacci: 0:00:00.642092
Time required to iterate and print the numbers: 0:00:00.000102
```

通过最后三行结果，我们会发现，代码中最费时的部分就是斐波那契数列的计算。



下载源代码

你可以用自己的账户登录http://www.packtpub.com，下载你购买过的Packt出版社的所有图书的示例代码。如果你是在其他地方购买的Packt出版社的书籍，可以通过http://www.packtpub.com/support注册账户，然后要求Packt把示例代码通过邮件发给你。

1.3.2　瓶颈在哪里

只要你测量出了程序的运行时间，就可以把注意力移到运行慢的环节上做性能分析。通常，瓶颈都是由下面的一种或几种原因造成的。

- 沉重的I/O操作，比如读取和分析大文件，长时间执行数据库查询，调用外部服务（比如HTTP请求），等等。
- 出现了内存泄漏，消耗了所有的内存，导致后面的程序没有内存来正常执行。
- 未经优化的代码被频繁地执行。
- 密集的操作在可以缓存时没有缓存，占用了大量资源。

I/O关联的代码（文件读/写、数据库查询等）很难优化，因为优化有可能会改变程序执行I/O操作的方式（通常是语言的核心函数操作I/O）。相反，优化计算关联的代码（比如程序使用的算法很糟糕），改善性能会比较容易（并不一定很简单）。这是因为优化计算关联的代码就是改写程序。

在性能优化接近尾声的时候，剩下的大多数性能瓶颈都是由I/O关联的代码造成的。

1.4　内存消耗和内存泄漏

软件开发过程中需要考虑的另一个重要资源是内存。一般的软件开发者不会意识到这一点，因为640KB RAM电脑的时代早已成为过去。但是一个内存泄漏的程序会把服务器糟蹋成640KB电脑。内存消耗不仅仅是关注程序使用了多少内存，还应该考虑控制程序使用内存的数量。

有一些开发项目，比如嵌入式系统开发，就会要求开发者关注内存配置，因为这类系统的资源是相当有限的。但是，普通开发者总希望目标系统能够提供他们需要的RAM。

随着RAM和高级编程语言都开始支持自动内存管理（比如垃圾回收机制），开发者不需要关注内存优化了，系统会帮忙完成的。

跟踪程序内存的消耗情况比较简单。最基本的方法就是使用操作系统的任务管理器。它会显示很多信息，包括程序占用的内存数量或者占用总内存的百分比。任务管理器也是检查CPU时间

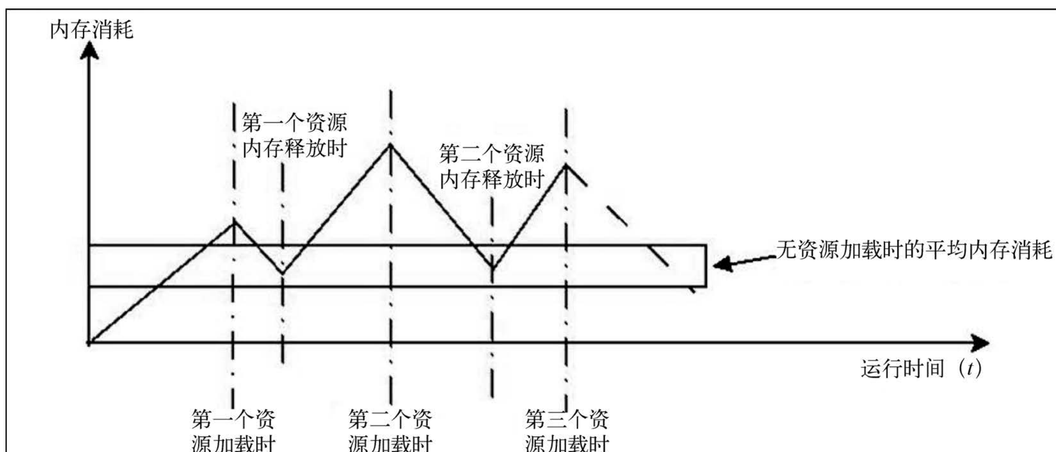
使用情况的好工具。在下面的截图中，你会发现一个简单的Python程序（就是前面那段程序）几乎占用了全部CPU（99.8%），内存只用了0.1%。

```
Activities                                                              Wed 23:48
top - 23:48:31 up 7 days, 14:22,  1 user,  load average: 0.63, 0.48, 0.44
Tasks: 316 total,   2 running, 313 sleeping,   1 stopped,   0 zombie
%Cpu(s): 15.2 us,  0.8 sy,  0.0 ni, 78.3 id,  5.8 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   7945412 total,  7722940 used,   222472 free,    27616 buffers
KiB Swap:  8155132 total,  2503856 used,  5651276 free.  1587528 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12469 fernando  20   0   40228   5564   2140 R 100.0  0.1   0:04.22 python
 3661 fernando  20   0 2124352 252696  21332 S   9.0  3.2  83:27.11 gnome-shell
 2585 root      20   0  873972 230080 198948 S   5.3  2.9  61:26.63 Xorg
 4483 fernando  20   0 3210628 1.328g  24340 S   5.3 17.5 414:52.19 firefox
 4193 fernando  20   0  792124  91796  15036 S   2.7  1.2  98:23.23 skype
10458 fernando  20   0 1041804 305700  67784 S   2.3  3.8   7:43.42 chrome
 4073 fernando  20   0  771384  19284   6372 S   1.3  0.2 129:36.18 chrome
 3541 fernando  20   0  396220  47876   1248 S   0.7  0.6   7:34.36 ibus-daemon
 4049 fernando  20   0  771384  19156   6308 S   0.7  0.2 129:06.05 chrome
 3625 fernando  20   0  200952   1232    696 S   0.3  0.0   1:19.11 ibus-engine-sim
 3640 fernando  20   0  446296   4412   2576 S   0.3  0.1  83:21.54 pulseaudio
 3742 fernando  20   0  781660  11172   5616 S   0.3  0.1   0:58.55 guake
 3985 fernando  20   0 2053884 304428  41080 S   0.3  3.8 144:36.67 chrome
 4027 fernando  20   0 2772532 618964 124452 S   0.3  7.8  54:11.44 chrome
 4685 fernando  20   0  957004  48140   4028 S   0.3  0.6  70:33.78 plugin-containe
 4729 fernando  20   0  585360   2668   1412 S   0.3  0.0   8:43.31 GoogleTalkPlugi
11031 fernando  20   0  703200  25092   7212 S   0.3  0.3   0:23.71 chrome
    1 root      20   0   34012   2352    800 S   0.0  0.0   0:02.65 init
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.10 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.57 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S   0.0  0.0   2:44.79 rcu_sched
    8 root      20   0       0      0      0 S   0.0  0.0   1:06.13 rcuos/0
    9 root      20   0       0      0      0 S   0.0  0.0   0:25.07 rcuos/1
   10 root      20   0       0      0      0 S   0.0  0.0   1:02.09 rcuos/2
   11 root      20   0       0      0      0 S   0.0  0.0   0:23.90 rcuos/3
   12 root      20   0       0      0      0 S   0.0  0.0   0:53.74 rcuos/4
   13 root      20   0       0      0      0 S   0.0  0.0   0:21.49 rcuos/5
   14 root      20   0       0      0      0 S   0.0  0.0   0:43.98 rcuos/6
   15 root      20   0       0      0      0 S   0.0  0.0   0:27.86 rcuos/7
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   17 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/0
```

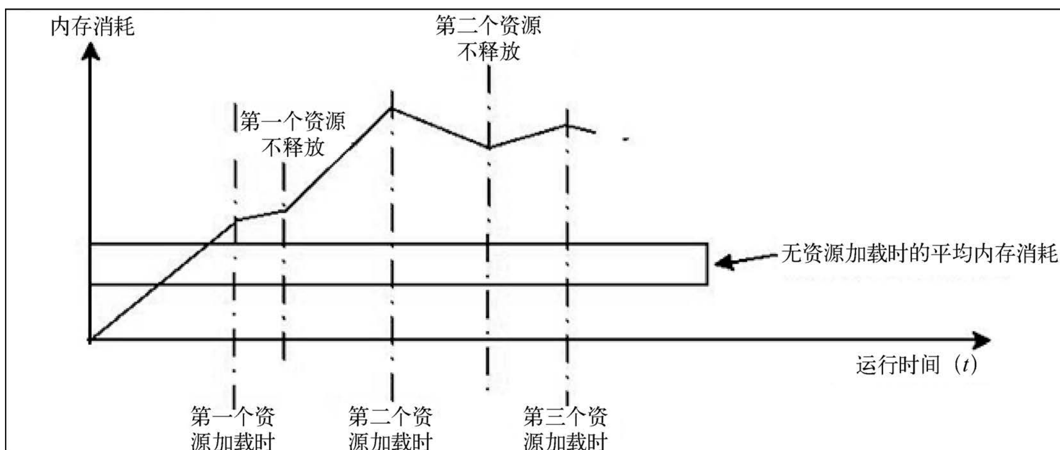
用这样的工具（Linux系统在命令行用`top`命令），可以轻松检测内存泄漏问题，不过这也要根据程序的具体情况综合考虑。如果你的程序在持续加载数据，那么其内存消耗的比例，可能会与那些没有频繁使用外部资源的程序不同。

例如，如果我们把一个调用大量外部资源的程序的内存消耗随时间的变化描绘出来，可能如下图所示。



资源加载时，内存使用曲线出现高峰；资源释放时，曲线会下降。虽然程序的内存消耗变化有点儿大，但是我们可以统计没有加载资源时程序的内存消耗的平均值。只要确定了这个平均值（图中用矩形表示），就可以判断内存泄漏的情况。

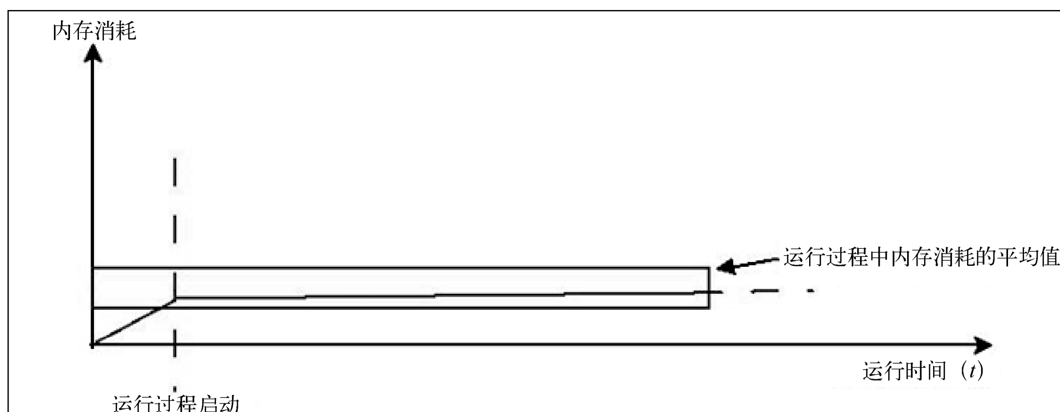
让我们再看一个资源加载效果比较糟糕的程序的内存消耗图（没有完全释放资源）。



在上图中你会发现，每当资源不再使用时，占用的内存并没有完全释放，这时内存消耗曲线就会位于矩形之上。这就表示程序会消耗越来越多的内存，即使加载资源已经释放也是如此。

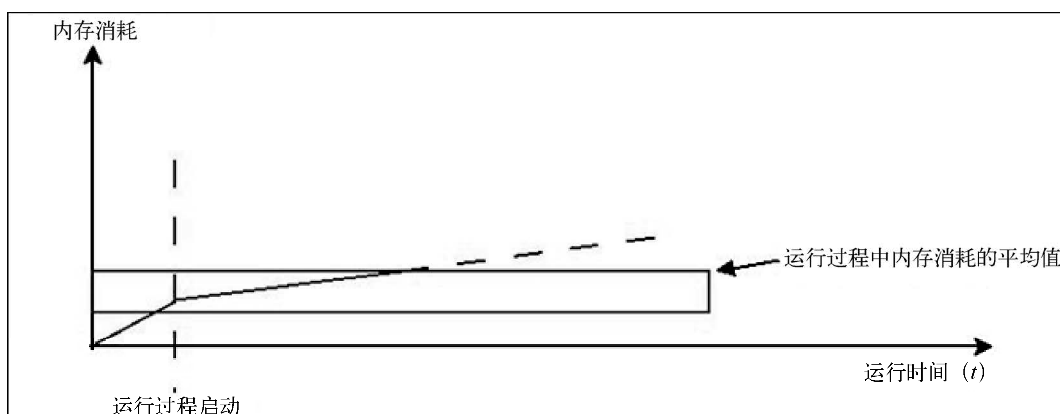
同理，也可以检测那些没有负载的程序的内存消耗情况，把执行特定任务的程序运行一段时间。有了数据，就很容易检测内存消耗和内存泄漏情况了。

让我们来看一个例子：



当运行过程启动之后，内存消耗会在一个范围内不断增加。如果发现增幅超出范围，而且消耗增大之后一直没有回落，就可以判断出现内存泄漏了。

一个内存泄漏的例子如下图所示。



1.5 过早优化的风险

优化通常被认为是一个好习惯。但是，如果一味优化反而违背了软件的设计原则就不好了。在开始开发一个新软件时，开发者经常犯的错误就是过早优化（permature optimization）。

如果过早优化代码，结果可能会和原来的代码截然不同。它可能只是完整解决方案的一部分，还可能包含因优化驱动的设计决策而导致的错误。

一条经验法则是，如果你还没有对代码做过测量（性能分析），优化往往不是个好主意。首先，应该集中精力完成代码，然后通过性能分析发现真正的性能瓶颈，最后对代码进行优化。

1.6 运行时间复杂度

在进行性能分析和优化时，理解运行时间复杂度（Running Time Complexity，RTC）的知识，以及学习使用它们适当地优化代码十分重要。

RTC可以用来对算法的运行时间进行量化。它是对算法在一定数量输入条件下的运行时间进行数学近似的结果。因为是数学近似，所以我们可以用这些数值对算法进行分类。

RTC常用的表示方法是大O标记（big O notation）。数学上，大O标记用于表示包含无限项的函数的有限特征（类似于泰勒展开式）。如果把这个概念用于计算机科学，就可以把算法的运行时间描述成渐进的有限特征（数量级）。

也就是说，这种标记通过宽泛的估计，让我们了解算法在任意数量输入下的运行时间。但是它不能提供精确的时间值，需要对代码进行深入的分析才能获得。

前面说过，用这种标记方法可以对算法进行分类，下面就是常用的算法类型。

1.6.1 常数时间——$O(1)$

常数时间（constant time）是最简单的算法复杂度类型。这基本上表示我们的测量结果将是恒定值，算法运行时间不会随着输入的增加而增加。

运行时间为$O(1)$的代码示例如下所示。

- 判断一个数是奇数还是偶数：

```
if number % 2:
    odd = True
else:
    odd = False
```

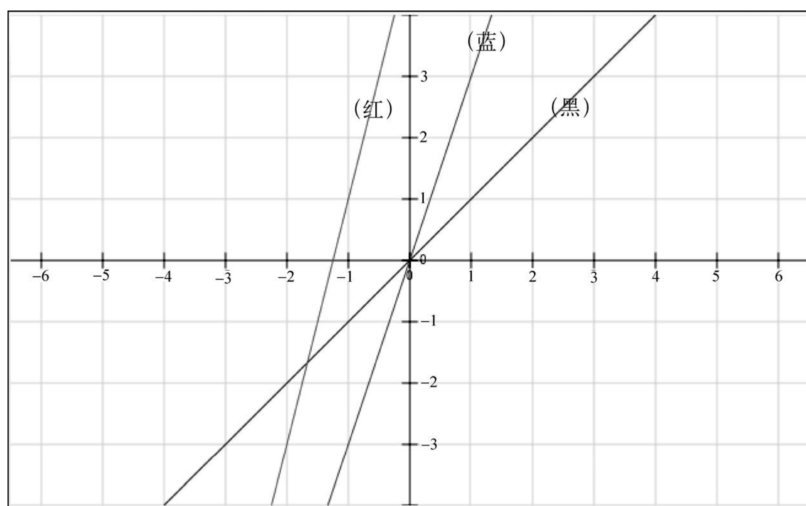
- 用标准输出方式打印信息：

```
print "Hello world!"
```

对于理论上更复杂的操作，比如在字典（或哈希表）中查找一个键的值，如果算法合理，就可以在常数时间内完成。技术上看，在哈希表中查找元素的消耗时间是$O(1)$平均时间，这意味着每次操作的平均时间（不考虑特殊情况）是固定值$O(1)$。

1.6.2 线性时间——$O(n)$

线性时间复杂度表示，在任意n个输入下，算法的运行时间与n呈线性关系，例如，$3n$，$4n+5$，等等。



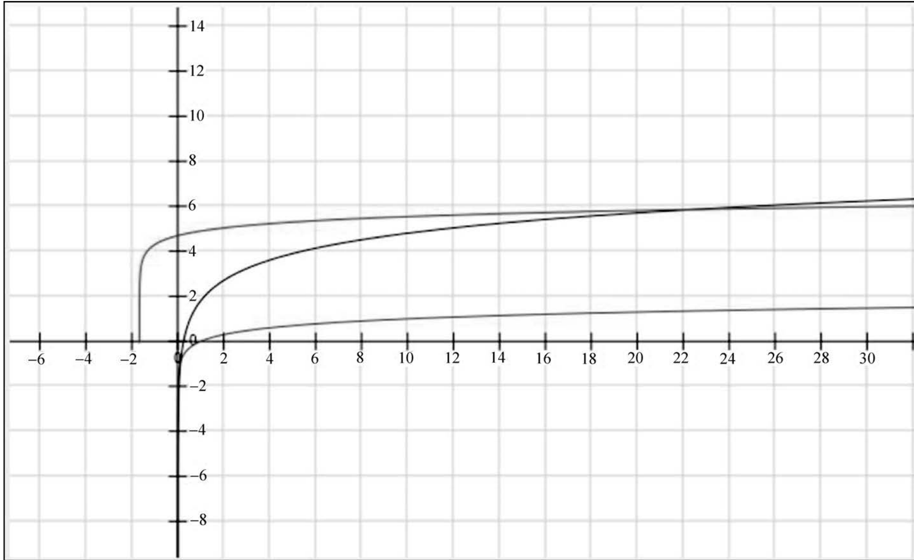
如上图所示，当x轴无线延伸时，蓝线（$3n$）和红线（$4n$+5）会和黑线（n）达到同样的上限。因此，为了简化，我们把这些算法都看成$O(n)$类。

这种数量级（order）的算法案例有：

- 查找无序列表中的最小元素
- 比较两个字符串
- 删除链表中的最后一项

1.6.3 对数时间——$O(\log n)$

对数时间（logarithmic time）复杂度的算法，表示随着输入数量的增加，算法的运行时间会达到固定的上限。随着输入数量的增加，对数函数开始增长很快，然后慢慢减速。它不会停止增长，但是越往后增长的速度越慢，甚至可以忽略不计。



上图显示了三种不同的对数函数。你会看到三条线都是同样的形状，随着x的增大，都是无限增加的。

对数时间的算法示例如下所示：

- 二分查找（binary search）
- 计算斐波那契数列（用矩阵乘法）

1.6.4　线性对数时间——$O(n\log n)$

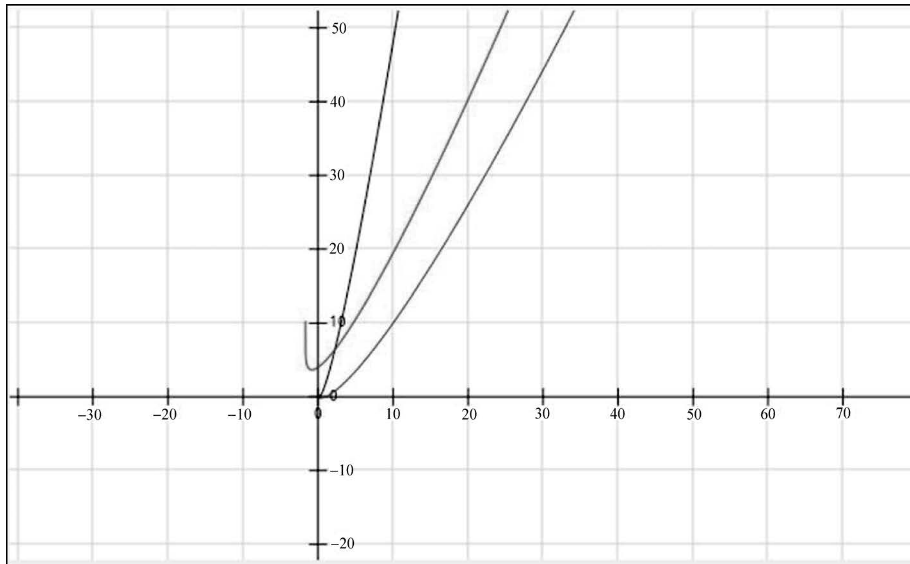
把前面两种时间类型组合起来就变成了线性对数时间（linearithmic time）。随着x的增大，算法的运行时间会快速增长。

这类算法的示例如下所示：

- 归并排序（merge sort）
- 堆排序（heap sort）
- 快速排序（quick sort，至少是平均运行时间）

下图中的线性对数函数曲线可以让我们更好地理解这类算法。

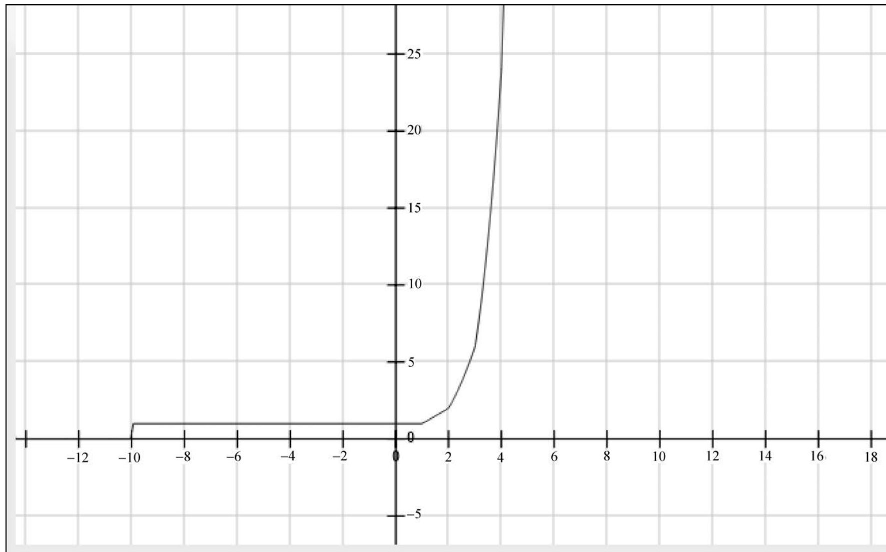
径）。



1.6.5 阶乘时间——$O(n!)$

阶乘时间（factorial time）复杂度的算法是最差的算法。其时间增速特别快，图都很难画。

下图是对阶乘函数的近似描述，可以看成这类算法的运行时间。



阶乘时间复杂度的一个示例，就是用暴力破解搜索方法解货郎担问题（遍历所有可能的路径）。

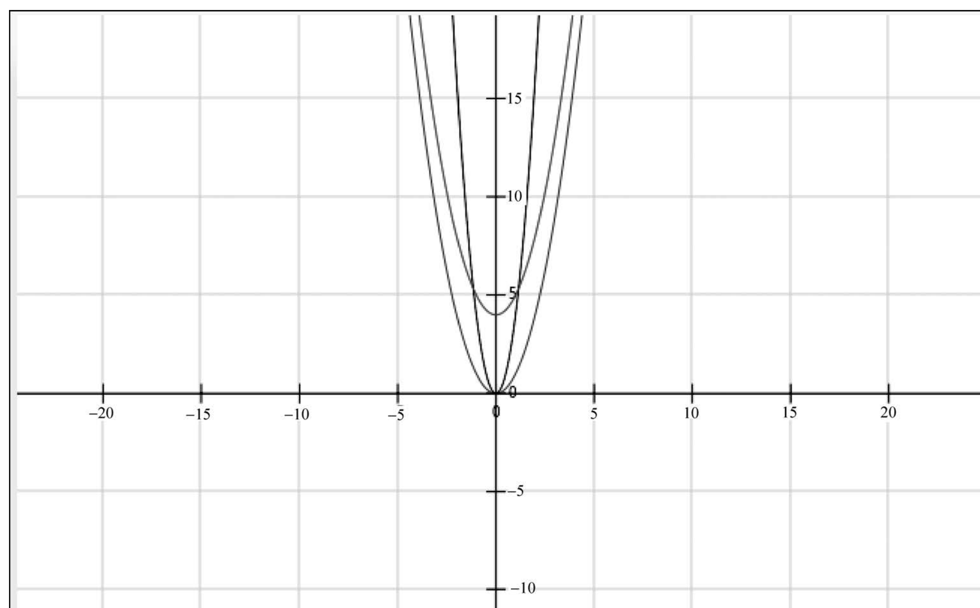
1.6.6　平方时间——$O(n^2)$

平方时间是另一个快速增长的时间复杂度。输入数量越多，需要消耗的时间越长（大多数算法都是这样，这类算法尤其如此）。平方时间复杂度的运行效率比线性时间复杂度要慢。

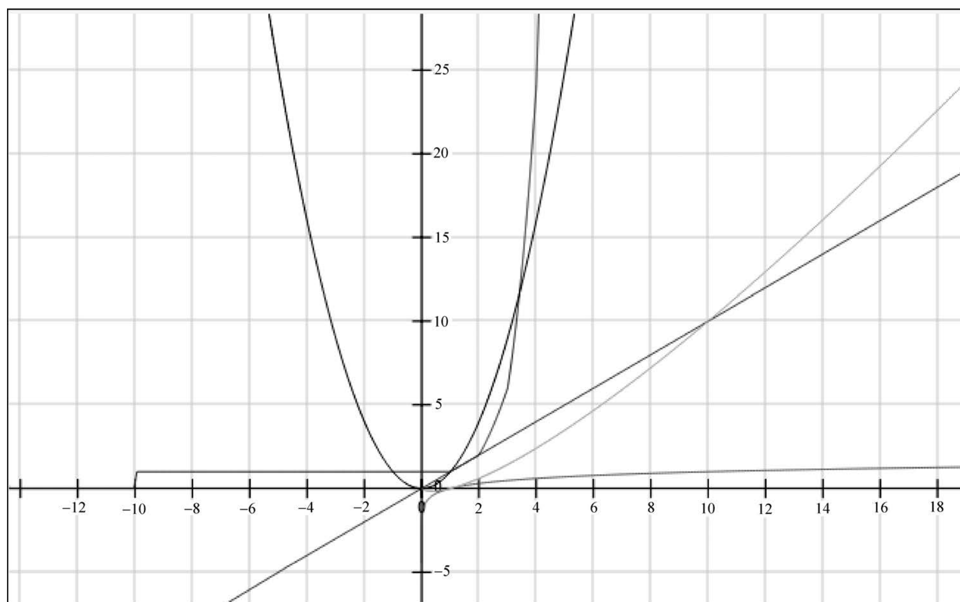
这类算法的示例如下：

- 冒泡排序（bubble sort）
- 遍历二维数组
- 插入排序（insertion sort）

这类函数的曲线图如下所示：



最后，我们把所有算法运行时间复杂度放在一张图上，比较一下运行效率：



不考虑常数时间复杂度（虽然它是最快的，但是显然复杂算法都不可能达到这个速度），那么时间复杂度排序如下所示：

- 对数
- 线性
- 线性对数
- 平方
- 阶乘

有时候，你可能也没办法，只能选择平方时间复杂度作为最佳解决方案。理论上我们总是希望实现更快速的算法，但是问题和技术的限制往往会影响结果。

> 注意，平方时间类型与阶乘时间类型之间，有一些变体，如三次方时间类型、四次方时间类型等。

还有很重要的一点需要考虑，就是算法的时间复杂度往往不止一种类型，可能是三种类型，包括最好情况、正常情况和最差情况。三种情况是由输入条件的不同属性决定的。例如，如果结果已经排序，插入排序算法的运行速度会比较快（最好情况），其他情况则要更慢一些（指数复杂度）。

另外数据类型也会影响时间复杂度。算法运行时间复杂度也与实际的操作方式有关（索引、插入、搜索等）。常见的数据类型和操作的时间复杂度如下所示。

数据结构	时间复杂度							
	正常情况				最差情况			
	索引	查找	插入	删除	索引	查找	插入	删除
列表（list）	$O(1)$	$O(n)$	–	–	$O(1)$	$O(n)$	–	–
单向链表（linked list）	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
双向链表（doubly linked list）	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
字典（dictionary）	–	$O(1)$	$O(1)$	$O(1)$	–	$O(n)$	$O(n)$	$O(n)$
二分查找树（Binary Search Tree，BST）	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

1.7　性能分析最佳实践

性能分析是重复性的工作。为了获得最佳性能，你可能需要在一个项目中做很多次性能分析，在另一个项目里还要再做一次。和软件开发中的其他重复性任务一样，有许多最佳实践可以帮助你高效地完成大多数性能分析工作。让我们来具体看看。

1.7.1　建立回归测试套件

在进行性能优化时，需要保证不管代码怎么变化，功能都不会变糟。最好的做法，尤其是面对大型项目时，就是建立测试套件。确保代码具有足够的覆盖率，可以让你信心去优化。覆盖率只有60%的测试套件在优化时可能会导致严重后果。

回归测试套件可以保证你在代码中尝试任何优化时，都不用担心代码的结构被破坏。

1.7.2　思考代码结构

函数代码之所以容易进行重构（refactor），是因为这种代码结构没有副作用。这样可以降低改变系统中其他部分的风险。如果你的代码没有局部可变的状态，将是另一个优势。这是因为，代码应该很容易理解和改变。没有按照前面的规则编写的代码，在重构过程中可能都需要额外的工作和注意。

1.7.3　耐心

性能分析不是一个快速、简单、精确的过程。也就是说，你不能指望运行一下性能分线器就可以把问题找到。有时候也许可以这样。但是，大多数情况下，你遇到的问题都不是很容易解决的。这就表明你必须浏览数据，描绘图形以便理解，不断地缩小检测范围，直到你重新开启新一轮分析，或者最终找到问题所在。

值得注意的是，对数据分析得越深入，表明你陷入的坑越深，数据将无法指明正确的优化方向，因此要时刻清楚自己的目标，并且在你开始之前已准备好正确的工具。然而，也可能搞了半天除了备受挫折，什么进展也没有。

1.7.4 尽可能多地收集数据

根据软件的不同类型和规模，在分析之前，你可能需要获取尽量多的数据。性能分析器很适合做这件事。但是，还有其他数据资源，如网络应用的系统日志、自定义日志、系统资源快照（如操作系统任务管理器），等等。

1.7.5 数据预处理

当你拥有了性能分析器的信息、日志和其他资源之后，在分析之前可能需要对数据进行预处理。不要因为性能分析器不能理解就回避非结构化数据。数据分析会往往从其他数据中受益。

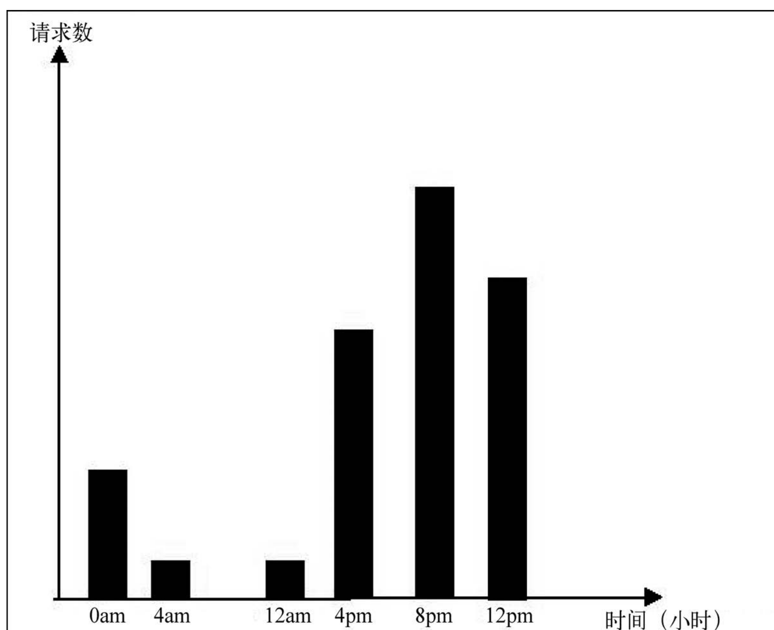
例如，如果分析网络应用的性能，获取网络服务器日志是个不错的主意，但是这些日志文件就是一行一个请求。解析文件并把数据存入数据库系统（像MongoDB、MySQL等），你就可以为数据确定含义（解析日期数据，通过IP获溯源地理位置等），并在后面进行查询。

前面这个过程称为ETL（extracting the data from it’s sources, transforming it into something with meaning, and loading it into another system），表示从源抽取数据，根据数据含义转换形式，并加载到其他系统中使用。

1.7.6 数据可视化

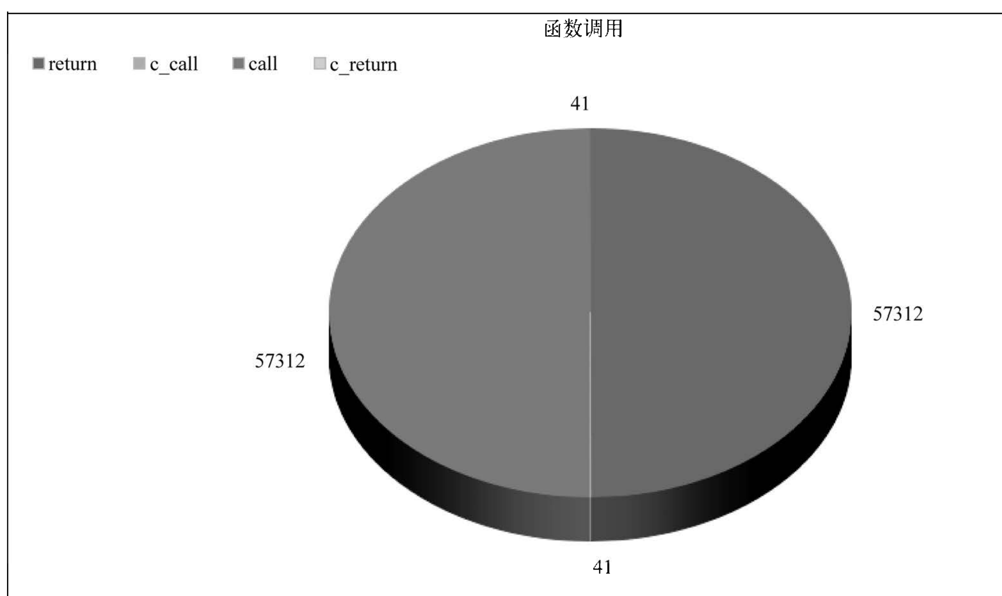
如果在错误发生之前，你不清楚自己要找的问题，只是想知道优化代码的方式，那么洞察你已经预处理过的数据的最好方式就是数据可视化。计算机很擅长处理数据，但是人类擅于通过图像来发现模式和理解现有信息中的某种特征。

例如，继续前面的网络服务器日志示例，一个简单的请求时间图（比如在微软的Excel中绘制）就可以显示客户行为的某种特征：



上图很清晰地显示出客户访问集中在下午晚些时候，并持续到深夜。后面你可以进一步针对这个特征进行性能分析。例如，针对这种现象的优化方案，可能就是在高峰期为基础设施增加更多资源（像亚马逊的AWS可以满足这类需求）。

另一个例子是用自定义性能分析数据可以画出下图：



上图是对本章第一个代码示例的性能分析结果中那些触发`profile`函数的事件进行数量统计。我们可以把它画成饼图，直观地看出数量最多的事件。可以看出，调用`call`和`return`占用了程序运行的绝大多数时间。

1.8　小结

在这一章，我们介绍了性能分析的基础知识，理解了性能分析方法及其重要性，并学会了如何使用它分析大多数代码的性能。

下一章我们将动手试试Python的性能分析器，看看它们是如何对应用进行性能分析的。

第2章 性能分析器

上一章介绍了性能分析的基础知识，展示了性能分析的重要性。如果把性能分析方法整合到开发过程中，就可以帮助我们提高产品的开发质量。另外，上一章还介绍了一些性能分析的具体方法。

上一章在最后介绍了程序运行时间复杂度的相关理论。在这一章，我们将会用到第一部分（关于性能分析的内容）。之后，我们将通过两个Python性能分析器（`cProfile`和`line_profiler`），把学到的理论付诸实践。

本章将介绍以下内容：

- 性能分析器的基本信息
- 性能分析器的下载和安装方法
- 通过示例演示性能分析器的功能
- 比较两种性能分析器的差异

2.1 认识新朋友：性能分析器

学完上一章所有的理论和简单示例之后，我们应该看看真正的Python了。我们先来看看目前最受关注也是用户最多的两个Python性能分析器：`cProfile`和`line_profiler`。两者将通过不同的方式帮助我们分析代码的性能。

`cProfile`（https://docs.python.org/2/library/profile.html#module-cProfile）从Python 2.5开始就是该语言默认的性能分析器，官方推荐在绝大多数场景中使用。而`line_profiler`（https://github.com/rkern/line_profiler）虽然不是Python官方发布的性能分析器，但是也被广泛使用。

下面详细地介绍两种性能分析器的相关知识。

2.2 cProfile

就像之前提到的，cProfile自Python 2.5以来就是标准版Python解释器默认的性能分析器。其他版本的Python，比如PyPy里面是没有cProfile的。它是一种确定性的性能分析器，提供了一组API帮助开发者收集Python程序运行的信息，更确切地说，是统计每个函数消耗的CPU时间。同时它还提供了其他细节，比如函数被调用的次数。

cProfile只测量CPU时间，并不关心内存消耗和其他与内存相关的信息统计。尽管如此，它是代码优化过程中一个很不错的起点，因为大多数时候，这个分析工具都会快速地为我们提供一组优化方案。

cProfile不需要安装，因为它是语言自带的一部分。要使用它，直接导入cProfile包即可。



确定性的性能分析器其实就是基于事件的性能分析器的另一种说法（更多细节请参考上一章内容）。也就是说，这个性能分析器会关注代码运行过程中的函数调用、返回语句等事件，甚至可以测量程序运行期间发生的每一个事件（与我们在上一章看到的统计式性能分析器不同）。

下面是从Python文档里提取出来的一个非常简单的例子：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

上面代码的输出结果如下：

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

从这个结果中可以收集到如下信息。

- 第一行告诉我们一共有197个函数调用被监控，其中192个是原生（primitive）调用，表明这些调用不涉及递归。
- ncalls表示函数调用的次数。如果在这一列中有两个数值，就表示有递归调用。第二个数值是原生调用的次数，第一个数值是总调用次数。这个数值可以帮助识别潜在的bug（当

数值大得异乎寻常时可能就是bug），或者可能需要进行内联函数扩展（inline expansion）的位置。

- tottime是函数内部消耗的总时间（不包括调用其他函数的时间）。这列信息可以帮助开发者找到可以进行优化的、运行时间较长的循环。
- percall是tottime除以ncalls，表示一个函数每次调用的平均消耗时间。
- cumtime是之前所有子函数消耗时间的累计和（也包含递归调用时间）。这个数值可以帮助开发者从整体视角识别性能问题，比如算法选择错误。
- 另一个percall是用cumtime除以原生调用的数量，表示到该函数调用时，每个原生调用的平均消耗时间。
- filename:lineno(function)显示了被分析函数所在的文件名、行号、函数名。

2.2.1　工具的局限

不存在透明的性能分析器。也就是说，即使像cProfile这样消耗极小的性能分析器，仍然会对代码实际的性能造成影响。当一个事件被触发时，事件实际发生的时间相比性能分析器查询到的系统内部时钟的时间，还是会有一些延迟。另外，当程序计数器离开性能分析器代码，回到用户代码中继续执行时，程序也会出现滞后。

除了这些之外，作为计算机内部的任何一段代码，内部时钟都有一个精度范围，任何小于这个精度的测量值都会被忽略。也就是说，如果进行性能分析的代码含有许多递归调用，或者一个函数需要调用许多函数，开发者就应该对这个函数做特殊处理，因为测量误差不断地累计最终会变得非常显著。

2.2.2　支持的 API

cProfile性能分析器提供了一些方法，帮助开发者收集程序中不同类型上下文的性能统计信息：

```
run(command, filename=None, sort=-1)
```

上面这个经典的方法，在前面的例子中使用过，用来收集命令执行的性能统计信息。当命令被调用时，会执行下面这个函数：

```
exec(command, __main__.__dict__, __main__.__dict__)
```

如果没有设置文件名filename，它就会创建一个新的stats类的实例（后面会详细介绍这个类）。下面的代码和之前的例子相同，但是带着新参数：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'stats', 'cumtime')
```

如果你运行这段代码，就会发现没有结果输出。但是，如果你检查文件夹里的内容，会发现一个新文件，叫stats。打开文件，你会发现里面的内容无法理解，因为文件是使用二进制格式保存的。后面将介绍如何读取文件信息并通过它来创建我们自己的报告：

```
runctx(command, globals, locals, filename=None)
```

这个方法和之前的很相似。唯一不同的是，它的参数列表中支持两个字典参数：globals和locals。当它被调用之后，会执行如下函数：

```
exec(command, globals, locals)
```

它会和run一样收集性能分析的统计信息。让我们用下面的例子来看看run和runctx之间的主要差别。

首先让我们用run函数，写出的代码如下：

```
import cProfile
def runRe():
    import re
    cProfile.run('re.compile("foo|bar")')
runRe()
```

当我们运行代码时，会得到下面的错误信息：

```
Traceback (most recent call last):
  File "cprof-test1.py", line 7, in <module>
      runRe() ...
  File "/usr/lib/python2.7/cProfile.py", line 140, in runctx
      exec cmd in globals, locals
  File "<string>", line 1, in <module>
NameError: name 're' is not defined
```

re模块没有被run方法发现，因为就像我们在前面见到的，run调用的exec函数的参数是__main__.__dict__。

现在，再让我们用runctx方法：

```
import cProfile
def runRe():
    import re
    cProfile.runctx('re.compile("foo|bar")', None, locals())
runRe()
```

它会输出下面的有效结果：

```
      195 function calls (190 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 re.py:192(compile)
     1    0.000    0.000    0.000    0.000 re.py:230(_compile)
```

```
        1    0.000    0.000    0.000    0.000 sre_compile.py:228(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:256(_optimize_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:433(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:546(isstring)
        1    0.000    0.000    0.000    0.000 sre_compile.py:552(_code)
        1    0.000    0.000    0.000    0.000 sre_compile.py:567(compile)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:64(_compile)
        5    0.000    0.000    0.000    0.000 sre_parse.py:137(__len__)
       12    0.000    0.000    0.000    0.000 sre_parse.py:141(__getitem__)
        7    0.000    0.000    0.000    0.000 sre_parse.py:149(append)
      3/1    0.000    0.000    0.000    0.000 sre_parse.py:151(getwidth)
        1    0.000    0.000    0.000    0.000 sre_parse.py:189(__init__)
       10    0.000    0.000    0.000    0.000 sre_parse.py:193(__next)
        2    0.000    0.000    0.000    0.000 sre_parse.py:206(match)
        8    0.000    0.000    0.000    0.000 sre_parse.py:212(get)
        1    0.000    0.000    0.000    0.000 sre_parse.py:317(_parse_sub)
        2    0.000    0.000    0.000    0.000 sre_parse.py:395(_parse)
        1    0.000    0.000    0.000    0.000 sre_parse.py:67(__init__)
        1    0.000    0.000    0.000    0.000 sre_parse.py:706(parse)
        3    0.000    0.000    0.000    0.000 sre_parse.py:92(__init__)
        1    0.000    0.000    0.000    0.000 {_sre.compile}
       15    0.000    0.000    0.000    0.000 {isinstance}
    39/38    0.000    0.000    0.000    0.000 {len}
        2    0.000    0.000    0.000    0.000 {max}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
                                               objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
        8    0.000    0.000    0.000    0.000 {min}
        6    0.000    0.000    0.000    0.000 {ord}
```

在性能分析过程中，`Profile(timer=None, timeunit=0.0, subcalls=True,builtins=True)`方法可以返回一个类，为开发者提供比`run`和`runctx`更多的控制。

`timer`参数是一个自定义函数，可以通过与默认函数不同的方式测量时间。它必须是一个可以返回当前时间数值的函数。如果开发者需要自定义函数，这个函数的消耗应该尽可能地低，避免测量口径造成的差异（请参考上一小节内容）。

如果`timer`的返回值是一个整数，那么`timeunit`参数就表示单位时间换算成秒的系数。例如，如果返回值单位时间是毫秒，那么`timeunit`应该就是`.001`。

让我们再看看`Profile`返回类的其他方法。

- `enable()`：表示开始收集性能分析数据。
- `disable()`：表示停止收集性能分析数据。
- `create_stats()`：表示停止收集数据，并为已收集的数据创建`stats`对象。
- `print_stats(sort=-1)`：创建一个`stats`对象，打印分析结果。
- `dump_stats(filename)`：把当前性能分析的内容写进一个文件。

- run(cmd)：和之前介绍过的run函数相同。
- runctx(cmd, globals, locals)：和之前介绍过的runctx函数相同。
- runcall(func, *args, **kwargs)：收集被调用函数func的性能分析信息。

让我们用下面的方式演示前面的例子：

```
import cProfile

def runRe():
    import re
    re.compile("foo|bar")

prof = cProfile.Profile()
prof.enable()
runRe()
prof.create_stats()
prof.print_stats()
```

在上面的性能分析代码中行数变多了，但这样做其实可以减少对原代码的干扰。通过这种方式对已经写好的代码或者已经通过测试的代码进行性能分析时，可以直接增加或删除性能分析代码，不需要调整源代码。

还有一种方式对源代码干扰更少，不需要增加任何代码，但是运行脚本时需要用一些命令行参数：

```
$ python -m cProfile your_script.py -o your_script.profile
```

需要注意的是，这样做会分析全部代码的性能，因此当你只想分析部分代码的性能时，这个方法可能无法返回想要的结果。

在介绍更详细、更有趣的例子之前，让我们再看看Stats类能为我们做什么。

2.2.3 Stats 类

pstats模块为开发者提供了Stats类，可以读取和操作stats文件（用之前介绍过的一种方法，把性能分析内容保存为二进制文件）的内容。

例如，下面的代码可以加载stats文件，打印里面经过排序的内容：

```
import pstats
p = pstats.Stats('stats')
p.strip_dirs().sort_stats(-1).print_stats()
```



注意Stats类的构造器可以接收cProfile.Profile类型的参数，可以不用文件名称作为数据源。

让我们更仔细地看看pstats.Stats类提供的方法。

- strip_dirs()：删除报告中所有函数文件名的路径信息。这个方法会改变stats实例内部的顺序，任何运行该方法的实例都将随机排列项目（每一行信息）的顺序。如果两个项目被认为是相同的（函数名相同，文件名相同，行数相同），那么这两个项目就可以合并。
- add(*filenames)：这个方法将文件名对应的文件的信息加载到当前的stats对象中。需要注意的是，和前面单独一个文件的处理方式相同，引用同一函数的stats项目（filename:lineno(function)，即文件名、行数和函数名）将被合并。
- dump_stats(filename)：就像cProfile.Profile类，这个方法把加载到Stats类的数据保存为一个文件。
- sort_stats(*keys)：这个方法从Python 2.3就开始出现，它是通过一系列条件依次对所有项目进行排序，从而调整stats对象的。当条件不止一个时，只有经过前一个条件排序后是相同的项目，才使用后一个条件进行排序。例如，如果sort_stats ('name', 'file')条件被使用，那么首先会把所有项目按照函数名排序，然后对名称相同的项目再按照文件名排序。

这个方法有时会自作聪明，遇到缩略词有歧义时就会自动按照拟定的规则进行理解，所以使用的时候要当心。目前支持排序的条件如下表所示。

准　　则	含　　义	升序/降序排列
calls	调用总次数	降序
cumulative	累计时间	降序
cumtime	累计时间	降序
file	文件名	升序
filename	文件名	升序
module	模块名	升序
ncalls	调用总次数	降序
pcalls	原始调用数	降序
line	行号	升序
name	函数名	升序
nfl	函数名/文件名/行号组合	降序
stdname	标准名称	升序
time	函数内部运行时间	降序
tottime	函数内部运行时间	降序



nfl与stdname

这两种排序方式的差异在于，stdname按照字符串打印的方式排序，就是把数字也作为字符串（4, 20, 30排序后就是20, 30, 4），而nfl是把行号字段作为数字进行排序。

最后，为了保持程序的兼容性，一些数字可以替换表格中的参数。-1、0、1、2分别表示stdname、calls、time、cumulative。

- reverse_order()：这个方法会逆反原来参数的排序方法（因此，如果原来按升序排列，现在就会变成降序）。
- print_stats(*restrictions)：这个方法是把信息打印到STDOUT。里面的可选参数用于体现打印结果的形式，可以是整数、小数和字符串。解释如下。
  - 整数：限制打印的行数。
  - 0.0到1.0（包含）之间的小数：表示按总行数的百分比打印。
  - 字符串：正则表达式，用于匹配stdname。

```
      196 function calls (191 primitive calls) in 0.000 seconds

   Random listing order was used
   List reduced from 34 to 10 due to restriction <10>
   List reduced from 10 to 6 due to restriction <'.*.py.*'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 /home/fernando/miniconda/lib/python2.7/sre_compile.py:567(compile)
        1    0.000    0.000    0.000    0.000 /home/fernando/miniconda/lib/python2.7/sre_compile.py:256(_optimize_charset)
      3/1    0.000    0.000    0.000    0.000 /home/fernando/miniconda/lib/python2.7/sre_parse.py:151(getwidth)
        1    0.000    0.000    0.000    0.000 /home/fernando/miniconda/lib/python2.7/sre_compile.py:552(_code)
        1    0.000    0.000    0.000    0.000 /home/fernando/miniconda/lib/python2.7/sre_compile.py:228(_compile_charset)
        2    0.000    0.000    0.000    0.000 /home/fernando/miniconda/lib/python2.7/sre_parse.py:206(match)
```

表示打印行数限制

上图显示的结果是从以下代码中调用print_stats输出的：

```
import cProfile
import pstats

def runRe():
    import re
    re.compile("foo|bar")
prof = cProfile.Profile()
prof.enable()
runRe()
prof.create_stats()

p = pstats.Stats(prof)
# 打印满足正则表达式的前10行结果
p.print_stats(10, 1.0, '.*.py.*')
```

如果函数里有多个参数，就依次满足各个参数。就像我们在上面那段代码里看到的，性能分析器的结果可能会非常长。但是，如果排序合理，就可以用参数汇总输出结果，获取需要的数据。

print_callers(*restrictions)方法的输入参数和使用规则与前面的函数相同，但是输出结果有一点儿不同。它会显示程序执行过程中调用的每个函数的调用次数、总时间和累计时间，以及文件名、行号和函数名的组合。

让我们通过下面的例子，看看如何通过cProfile.Profile和Stats获取程序调用函数列表：

```
import cProfile
import pstats

def runRe():
    import re
    re.compile("foo|bar")
prof = cProfile.Profile()
prof.enable()
runRe()
prof.create_stats()

p = pstats.Stats(prof)
p.print_callers()
```

注意观察这里是如何把`profile.Stats`和`cProfile.Profile`组合起来使用的。它们会同时运行并共同显示我们想要的结果。现在，让我们看看输出结果：

```
   Random listing order was used

Function                                                                   was called by...
                                                                               ncalls  tottime  cumtime
{_sre.compile}                                                             <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:567(compile)
/home/fernando/miniconda/lib/python2.7/sre_compile.py:567(compile)         <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/re.py:230(_compile)
/home/fernando/miniconda/lib/python2.7/sre_compile.py:256(_optimize_charset) <-     1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:228(_compile_charset)
{method 'find' of 'bytearray' objects}                                     <-       5    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:256(_optimize_charset)
{ord}                                                                      <-       6    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:395(_parse)
/home/fernando/miniconda/lib/python2.7/sre_parse.py:151(getwidth)          <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:151(getwidth)
/home/fernando/miniconda/lib/python2.7/sre_compile.py:552(_code)           <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:567(compile)
/home/fernando/miniconda/lib/python2.7/sre_compile.py:228(_compile_charset) <-      1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)
{min}                                                                      <-       8    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:151(getwidth)
/home/fernando/miniconda/lib/python2.7/sre_parse.py:206(match)             <-       2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:317(_parse_sub)
/home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)   <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:552(_code)
{len}                                                                      <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/re.py:230(_compile)
                                                                                    8    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:64(_compile)
                                                                                    4    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:256(_optimize_charset)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)
                                                                                    5    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:137(__len__)
                                                                                   17    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:193(__next)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:317(_parse_sub)
/home/fernando/miniconda/lib/python2.7/sre_parse.py:137(__len__)           <-       2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:317(_parse_sub)
                                                                                    1    0.000    0.000  {len}
/home/fernando/miniconda/lib/python2.7/sre_parse.py:141(__getitem__)       <-       8    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:64(_compile)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:317(_parse_sub)
/home/fernando/miniconda/lib/python2.7/sre_parse.py:189(__init__)          <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:706(parse)
/home/fernando/miniconda/lib/python2.7/sre_parse.py:395(_parse)            <-       2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:317(_parse_sub)
/home/fernando/miniconda/lib/python2.7/cProfile.py:90(create_stats)        <-
/home/fernando/miniconda/lib/python2.7/sre_parse.py:67(__init__)           <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:706(parse)
/home/fernando/miniconda/lib/python2.7/re.py:192(compile)                  <-       1    0.000    0.000  B02088_02_07.py:4(runRe)
{isinstance}                                                               <-       1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/re.py:230(_compile)
                                                                                    2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:546(isstring)
```

`print_callees(*restrictions)`方法打印一列调用其他函数的函数。其数据显示格式和限制参数与上一个函数相同。

你可能会看到部分输出结果如下面的截图所示：

```
{method 'append' of 'list' objects}          <-      22    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:64(_compile)
                                                      5    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:228(_compile_charset)
                                                      4    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:256(_optimize_charset)
                                                      7    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:433(_compile_info)
                                                      1    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_compile.py:552(_code)
                                                      7    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:149(append)
                                                      2    0.000    0.000  /home/fernando/miniconda/lib/python2.7/sre_parse.py:317(_parse_sub)
```

这个结果表示右边的函数是被左边的函数调用的。

2.2.4　性能分析示例

现在我们已经掌握了`cProfile`和`Stats`的基本用法了，下面来探索一些更加有趣且真实的

例子吧。

1. 回到斐波那契数列

让我们重新回到斐波那契数列，因为用递归方式计算的斐波那契数列有很大的改进空间。

让我们先看看未经性能分析也没有优化过的代码：

```
import profile

def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

profile.run('print fib_seq(20); print')
```

代码的输出结果如下：

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]

         57356 function calls (66 primitive calls) in 0.142 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       21    0.001    0.000    0.001    0.000 :0(append)
       20    0.000    0.000    0.000    0.000 :0(extend)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.142    0.142 <string>:1(<module>)
 57291/21    0.141    0.000    0.141    0.007 B02088_02_08.py:3(fib)
     21/1    0.000    0.000    0.142    0.142 B02088_02_08.py:9(fib_seq)
        1    0.000    0.000    0.142    0.142 profile:0(print fib_seq(20); print)
        0    0.000             0.000          profile:0(profiler)
```

虽然输出的结果打印没问题，但是看看上图中画框的部分。具体解释如下：

- 在0.142秒内，共有57 356个函数调用
- 一共只有66个原生调用（不包括递归）
- 在代码的第三行，一共有57 270（57 291–21）次递归函数调用

我们已经知道，过多的函数调用将增加额外的时间消耗。从图中可见（`cumtime`列），大部分时间都消耗在递归函数里了，因此我们有理由确信如果让递归函数加速，整个程序的执行时间也会改善。

现在，让我们给fib函数加一个简单的装饰器，缓存之前计算的值［这是一种函数返回值缓存（memoization）技术，将在后面的章节里介绍］，这样每个fib函数的值就不需要重复计算了：

```
import profile

class cached:
    def __init__(self, fn):
        self.fn = fn
        self.cache = {}

    def __call__(self, *args):
        try:
            return self.cache[args]
        except KeyError:
            self.cache[args] = self.fn(*args)
            return self.cache[args]

@cached
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

profile.run('print fib_seq(20); print')
```

现在，让我们运行代码，结果如下所示：

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]

         145 function calls (87 primitive calls) in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       21    0.000    0.000    0.000    0.000 :0(append)
       20    0.000    0.000    0.000    0.000 :0(extend)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
       21    0.000    0.000    0.000    0.000 B02088_02_09.py:15(fib)
     21/1    0.000    0.000    0.001    0.001 B02088_02_09.py:22(fib_seq)
    59/21    0.000    0.000    0.000    0.000 B02088_02_09.py:8(__call__)
        1    0.000    0.000    0.001    0.001 profile:0(print fib_seq(20); print)
        0    0.000             0.000          profile:0(profiler)
```

程序的函数调用次数从57 000多下降到了145，运行时间也从0.142秒下降到了0.001秒。这是一个非常给力的优化！但是，我们的原生调用变多了，而递归调用明显减少了。

让我们再进行另一项优化。虽然我们的例子对单个函数调用的处理非常快，但是让我们试试

把多个调用合成一组，用stats输出结果。这样做可能还会看到一些有趣的新结果。为此，我们需要使用stats模块。示例代码如下：

```
import cProfile
import pstats
from fibo4 import fib, fib_seq

filenames = []
profiler = cProfile.Profile()
profiler.enable()
for i in range(5):
    print fib_seq(1000); print
profiler.create_stats()
stats = pstats.Stats(profiler)
stats.strip_dirs().sort_stats('cumulative').print_stats()
stats.print_callers()
```

我们已经做了简化。计算1000个斐波那契数列可能计算量太大，尤其是递归实现时。这样运行代码肯定会超过cPython的递归限制。cPython为了防止栈溢出，设置了递归保护措施（理论上，这个问题可以通过尾递归解决，但是cPython没有提供）。因此，我们找到了另一个解决方案。让我们修复这个问题，运行下面的代码：

```
import profile

def fib(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a+b
    return a

def fib_seq(n):
    seq = []
    for i in range(0, n + 1):
        seq.append(fib(i))
    return seq

print fib_seq(1000)
```

上面的代码会打印出存储很多数值的超长列表，但是这些行证明我们实现了目标。我们可以计算1000个斐波那契数列。现在，让我们分析看看结果如何。

用新的性能分析函数，不过需要迭代版的斐波那契实现，代码如下：

```
import cProfile
import pstats
from fibo_iter import fib, fib_seq

filenames = []
profiler = cProfile.Profile()
profiler.enable()
for i in range(5):
```

```
    print fib_seq(1000); print
profiler.create_stats()
stats = pstats.Stats(profiler)
stats.strip_dirs().sort_stats('cumulative').print_stats()
stats.print_callers()
```

这段代码显示在命令行的结果如下：

```
         15028 function calls in 0.187 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.002    0.000    0.187    0.037 fibo_iter.py:10(fib_seq)
     5005    0.173    0.000    0.184    0.000 fibo_iter.py:3(fib)
     5011    0.011    0.000    0.011    0.000 {range}
     5005    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 cProfile.py:90(create_stats)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


   Ordered by: cumulative time

Function                                          was called by...
                                                      ncalls  tottime  cumtime
fibo_iter.py:10(fib_seq)                          <-
fibo_iter.py:3(fib)                               <-    5005    0.173    0.184  fibo_iter.py:10(fib_seq)
{range}                                           <-    5005    0.011    0.011  fibo_iter.py:3(fib)
                                                           5    0.000    0.000  fibo_iter.py:10(fib_seq)
{method 'append' of 'list' objects}               <-    5005    0.001    0.001  fibo_iter.py:10(fib_seq)
cProfile.py:90(create_stats)                      <-
{method 'disable' of '_lsprof.Profiler' objects}  <-       1    0.000    0.000  cProfile.py:90(create_stats)
```

新代码用0.187秒计算了1000个斐波那契数列5次。虽然这个结果并不差，但是我们知道可以用缓存数值来优化，和我们之前做的一样。你已经看到，`fib`函数调用了5005次，如果缓存结果，就可以大幅度降低调用次数，这意味着很少的运行时间。

只需要一点点努力，我们就可以通过缓存改善之前被调用了5005次的`fib`函数的调用时间：

```
import profile


class cached:
    def __init__(self, fn):
        self.fn = fn
        self.cache = {}

    def __call__(self, *args):
        try:
            return self.cache[args]
        except KeyError:
            self.cache[args] = self.fn(*args)
            return self.cache[args]


@cached
def fib(n):
    a, b = 0, 1
    for i in range(0, n):
```

```
        a, b = b, a+b
    return a


def fib_seq(n):
    seq = []
    for i in range(0, n + 1):
        seq.append(fib(i))
    return seq

print fib_seq(1000)
```

你应该会得到如下的运行结果：

```
        10023 function calls in 0.006 seconds

  Ordered by: cumulative time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       5    0.004    0.001    0.006    0.001 fibo_iter.py:25(fib_seq)
    5005    0.002    0.000    0.002    0.000 fibo_iter.py:8(__call__)
    5005    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
       6    0.000    0.000    0.000    0.000 {range}
       1    0.000    0.000    0.000    0.000 cProfile.py:90(create_stats)
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


  Ordered by: cumulative time

Function                                          was called by...
                                                      ncalls  tottime  cumtime
fibo_iter.py:25(fib_seq)                          <-
fibo_iter.py:8(__call__)                          <-    5005    0.002    0.002  fibo_iter.py:25(fib_seq)
{method 'append' of 'list' objects}               <-    5005    0.000    0.000  fibo_iter.py:25(fib_seq)
{range}                                           <-       5    0.000    0.000  fibo_iter.py:25(fib_seq)
cProfile.py:90(create_stats)                      <-
{method 'disable' of '_lsprof.Profiler' objects}  <-       1    0.000    0.000  cProfile.py:90(create_stats)
```

只要简单地缓存`fib`函数的结果，就可以把运行时间从0.187秒缩短到0.006秒。这是非常给力的优化！干得漂亮！

2. 推文数据统计

让我们再看一个内容上更复杂一些的例子，毕竟斐波那契数列不是现实中人人都会用到的。还是让我们再做一些有趣的事情吧。现在Twitter已经允许你以CSV格式下载自己的推文列表。我们就用自己的数据做一些统计。

通过获取的数据，我们可以统计下面的信息：

- 实际回复的信息占比
- 网站（https://twitter.com）发布的推文占比
- 手机发布的推文占比

我们的程序输出的结果应该如下图所示。

```
-------- My twitter stats ------------
35% of tweets are replies
86% of tweets were made from the website
13% of tweets were made from my phone
```

为了简便，我们重点关注CVS文件解析，并做一些基本计算。我们不用任何第三方模块，这样，我们就可以完全控制代码和分析的内容了。也就是说，像Python的`csv`模块我们也不用。

前面出现过的其他不太好的做法，比如`inc_stat`函数，或者在处理文件之前把整个文件都载入内存，都会提醒你，这只是一个显示基本改进方法的示例。

下面是初始代码：

```
def build_twit_stats():
    STATS_FILE = './files/tweets.csv'
    STATE = {
        'replies': 0,
        'from_web': 0,
        'from_phone': 0,
        'lines_parts': [],
        'total_tweets': 0
    }
    read_data(STATE, STATS_FILE)
    get_stats(STATE)
    print_results(STATE)

def get_percentage(n, total):
    return (n * 100) / total

def read_data(state, source):
    f = open(source, 'r')

    lines = f.read().strip().split("\"\n\"")
    for line in lines:
       state['lines_parts'].append(line.strip().split(','))
    state['total_tweets'] = len(lines)

def inc_stat(state, st):
    state[st] += 1

def get_stats(state):
    for i in state['lines_parts']:
        if(i[1] != '""'):
            inc_stat(state, 'replies')
        if(i[4].find('Twitter Web Client') > -1):
            inc_stat(state, 'from_web')
        else:
            inc_stat(state, 'from_phone')

def print_results(state):
    print "-------- My twitter stats -------------"
    print "%s%% of tweets are replies" % (get_percentage(state['replies'], state
        ['total_tweets']))
```

```
    print "%s%% of tweets were made from the website" % (get_percentage(state
        ['from_web'], state['total_tweets']))
    print "%s%% of tweets were made from my phone" % (get_percentage(state['from_
        phone'], state['total_tweets']))
```

其实，这段代码并不是太复杂，就是读取文件内容，按行分割，再把每一行分配到不同的类型中，最后统计各个类型推文的数量。初看这段代码，可能会认为没什么可优化的，但我们会发现其实还是有优化空间的。

另一个需要注意的地方是，我们要处理的数据有150MB。

下面的代码会导入代码并使用它生成性能分析报告：

```
import cProfile
import pstats
from B02088_02_14 import build_twit_stats


profiler = cProfile.Profile()
profiler.enable()

build_twit_stats()

profiler.create_stats()
stats = pstats.Stats(profiler)
stats.strip_dirs().sort_stats('cumulative').print_stats()
```

执行代码获得的结果如下：

```
-------- My twitter stats -------------
34% of tweets are replies
86% of tweets were made from the website
13% of tweets were made from my phone
         3019962 function calls in 2.059 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.026    0.026    2.059    2.059 B02088_02_14.py:6(build_twit_stats)
        1    0.262    0.262    1.582    1.582 B02088_02_14.py:22(read_data)
   564851    1.031    0.000    1.031    0.000 {method 'split' of 'str' objects}
        1    0.257    0.257    0.450    0.450 B02088_02_14.py:33(get_stats)
   564851    0.226    0.000    0.226    0.000 {method 'strip' of 'str' objects}
   760548    0.099    0.000    0.099    0.000 B02088_02_14.py:30(inc_stat)
   564850    0.095    0.000    0.095    0.000 {method 'find' of 'str' objects}
        1    0.039    0.039    0.039    0.039 {method 'read' of 'file' objects}
   564850    0.024    0.000    0.024    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 B02088_02_14.py:42(print_results)
        1    0.000    0.000    0.000    0.000 {open}
        1    0.000    0.000    0.000    0.000 cProfile.py:90(create_stats)
        1    0.000    0.000    0.000    0.000 {len}
        3    0.000    0.000    0.000    0.000 B02088_02_14.py:19(get_percentage)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

上面的截屏中有三点需要注意：

(1) 程序的总执行时间

(2) 不同函数累计的调用次数

(3) 每个函数的总调用次数

我们的目标是减少总执行时间。因此，我们需要考虑不同函数调用的累计次数和每个函数的总调用次数。关于这两点，我们可以得出下面的结论。

- build_twit_stats消耗了最多的时间。然而，你会看到在前面的代码中，它只是调用了其他所有函数，这是显而易见的。我们可以把注意力集中在耗时第二多的read_data函数上。这倒是挺有意思，我们的性能瓶颈不是计算统计数据，而是读取文件。
- 在代码的第三行，我们可以清楚地看到read_data函数的瓶颈。我们使用了太多split命令，它们的时间累加了。
- 还可以看到第四耗时的函数get_stats。

那么现在让我们带着这些问题，看看有没有更好的解决方案。最大的性能瓶颈就是加载数据。我们首先把数据加载到内存中，然后重复地遍历文件计算统计数据。我们可以改成逐行读取文件，然后每读一行统计一次。让我们看看代码应该怎么写。

新的read_data函数应该像这样：

```
def read_data(state, source):
    f = open(source)

    buffer_parts = []
    for line in f:
      # 由于多行推文在文件中也被保存为若干行，
      # 因此需要考虑把它们合并到一起。
      parts = line.split('","')
      buffer_parts += parts
      if len(parts) == 10:
        state['lines_parts'].append(buffer_parts)
        get_line_stats(state, buffer_parts)
        buffer_parts = []
    state['total_tweets'] = len(state['lines_parts'])
```

我们需要增加一些逻辑处理多行推文，也就是CSV文件中的多行记录。我们把get_stats函数改成了get_line_stats，这样做可以简化逻辑，因为它只计算当前行的值。

```
def get_line_stats(state, line_parts):
    if line_parts[1] != '' :
        state['replies'] += 1
    if 'Twitter Web Client' in line_parts[4]:
        state['from_web'] += 1
    else:
        state['from_phone'] += 1
```

最后两项改进，一是移除inc_stat函数的调用，由于我们用字典，所以调用就没必要了；二是利用in操作符替换find方法。

让我们再运行一次代码，看看变化：

```
-------- My twitter stats -------------
34% of tweets are replies
86% of tweets were made from the website
13% of tweets were made from my phone
         2312158 function calls in 1.590 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.590    1.590 B02088_02_16.py:3(build_twit_stats)
        1    0.607    0.607    1.589    1.589 B02088_02_16.py:19(read_data)
   604612    0.750    0.000    0.750    0.000 {method 'split' of 'str' objects}
   551462    0.173    0.000    0.173    0.000 B02088_02_16.py:38(get_line_stats)
   604613    0.033    0.000    0.033    0.000 {len}
   551462    0.027    0.000    0.027    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 B02088_02_16.py:46(print_results)
        3    0.000    0.000    0.000    0.000 B02088_02_16.py:16(get_percentage)
        1    0.000    0.000    0.000    0.000 {open}
        1    0.000    0.000    0.000    0.000 cProfile.py:90(create_stats)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

运行时间从2秒降到了1.6秒，算是一个不错的改进。read_data函数仍然是耗时最多的函数，但是现在的原因是get_line_stats函数。我们还可以进一步优化它，虽然这个函数并没有做很多操作，但是在循环体中不断地调用它会消耗一些查询时间。我们可以对这个函数进行内联，看看效果有没有改善。

新的代码如下：

```
def read_data(state, source):
    f = open(source)

    buffer_parts = []
    for line in f:
      # 由于多行推文在文件中也被保存为若干行，
      # 因此需要考虑把它们合并到一起。
      parts = line.split('","')
      buffer_parts += parts
      if len(parts) == 10:
        state['lines_parts'].append(buffer_parts)
        if buffer_parts[1] != '' :
          state['replies'] += 1
        if 'Twitter Web Client' in buffer_parts[4]:
          state['from_web'] += 1
        else:
          state['from_phone'] += 1
        buffer_parts = []
    state['total_tweets'] = len(state['lines_parts'])
```

现在，结果有了新变化，如下图所示。

```
-------- My twitter stats -------------
34% of tweets are replies
86% of tweets were made from the website
13% of tweets were made from my phone
         1760696 function calls in 1.423 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.423    1.423 B02088_02_16.py:3(build_twit_stats)
        1    0.624    0.624    1.423    1.423 B02088_02_16.py:19(read_data)
   604612    0.746    0.000    0.746    0.000 {method 'split' of 'str' objects}
   604613    0.028    0.000    0.028    0.000 {len}
   551462    0.025    0.000    0.025    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 B02088_02_16.py:40(print_results)
        1    0.000    0.000    0.000    0.000 cProfile.py:90(create_stats)
        1    0.000    0.000    0.000    0.000 {open}
        3    0.000    0.000    0.000    0.000 B02088_02_16.py:16(get_percentage)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

相比第一张图和前面那一张图，这是很明显的进步。我们把程序运行时间从2秒降到了1.4秒。函数调用次数也明显降低了（从大约300万次降到了170万次），相应地也减少了函数查询和调用的时间。

作为额外的改善，我们还可以简化代码以增加可读性。最终的代码如下：

```
def build_twit_stats():
    STATS_FILE = './files/tweets.csv'
    STATE = {
        'replies': 0,
        'from_web': 0,
        'from_phone': 0,
        'lines_parts': [],
        'total_tweets': 0
    }
    read_data(STATE, STATS_FILE)
    print_results(STATE)

def get_percentage(n, total):
    return (n * 100) / total

def read_data(state, source):
    f = open(source)

    buffer_parts = []
    for line in f:
      # 由于多行推文在文件中也被保存为若干行，
      # 因此需要考虑把它们合并到一起。
      parts = line.split('","')
      buffer_parts += parts
      if len(parts) == 10:
        state['lines_parts'].append(buffer_parts)
        if buffer_parts[1] != '' :
          state['replies'] += 1
        if 'Twitter Web Client' in buffer_parts[4]:
          state['from_web'] += 1
        else:
```

```
            state['from_phone'] += 1
          buffer_parts = []
      state['total_tweets'] = len(state['lines_parts'])


def print_results(state):
    print "-------- My twitter stats -------------"
    print "%s%% of tweets are replies" % (get_percentage(state['replies'],
state['total_tweets']))
    print "%s%% of tweets were made from the website" % (get_percentage(state
['from_web'], state['total_tweets']))
    print "%s%% of tweets were made from my phone" % (get_percentage(state
['from_phone'], state['total_tweets']))
```

下面我们对cProfile做个总结。通过它我们可以对代码进行性能分析，获取每个函数的调用次数和总调用次数。它帮助我们通过系统全局视角改进代码。下面将介绍另一种性能分析器，它可以为我们提供每一行代码的性能细节，这是cProfile无法提供的。

2.3 line_profiler

这个性能分析器和cProfile不同。它可以帮助你一行一行地分析函数性能，而不是像cProfile那样做确定性性能分析。

可以用pip（https://pypi.python.org/pypi）命令行工具，通过下面的代码安装line_profiler：

```
$ pip install line_profiler
```



如果安装过程中遇到问题，比如文件缺失，请确保你已经安装了相关依赖。在Ubuntu中，可以通过下面的命令安装需要的依赖：

```
$ sudo apt-get install python-dev libxml2-dev libxslt-dev
```

line_profiler试图弥补cProfile和类似性能分析器的不足。其他性能分析器主要关注函数调用消耗的CPU时间。大多数情况下，这足以发现问题，消除瓶颈（就像我们之前看到的那样）。但是，有时候，瓶颈问题发生在函数的某一行中，这时就需要line_profiler解决了。

line_profiler的作者建议使用kernprof工具，后面我们会介绍相关示例。kernprof会创建一个性能分析器实例，并把名字添加到__builtins__命名空间的profile中。line_profiler性能分析器被设计成一个装饰器，你可以装饰任何一个函数，它会统计每一行消耗的时间。

用下面的代码执行这个性能分析器：

```
$ kernprof -l script_to_profile.py
```

被装饰的函数将被分析：

```
@profile
def fib(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a+b
    return a
```

kernprof默认情况下会把分析结果写入script_to_profile.py.lprof文件，不过你可以用-v属性让结果立即显示在命令行里：

```
$ kernprof -l -v script_to_profile.py
```

下面是一个简单的示例结果，可帮助你理解看到的内容。

```
Wrote profile results to kernprof-test.py.lprof
Timer unit: 1e-06 s

Total time: 7.3e-05 s
File: kernprof-test.py
Function: test at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def test():
     4        11            5      0.5      6.8      for i in range(0, 10):
     5        10           63      6.3     86.3          print i**2
     6         1            5      5.0      6.8      print "End of the function"
```

结果会显示函数的每一行，旁边是时间信息。共有6列信息，具体含义如下。

- Line #：表示文件中的行号。
- Hits：性能分析时一行代码的执行次数。
- Time：一行代码执行的总时间，由计时器的单位决定。在分析结果的最开始有一行Timer unit，该数值就是转换成秒的计时单位（要计算总时间，需要用Time数值乘以计时单位）。不同系统的计时单位可能不同。
- Per hit：执行一行代码的平均消耗时间，依然由系统的计时单位决定。
- % Time：执行一行代码的时间消耗占程序总消耗时间的比例。

如果你正在使用line_profiler进行性能分析，有两种方式可以获得函数的性能分析数据：用构造器或者用add_function方法。

line_profiler和cProfile.Profile一样，也提供了run、runctx、runcall、enable和disable方法。但是最后两个函数在嵌入模块统计性能时并不安全，使用时要当心。进行性能分析之后，可以用dump_stats(filename)方法把stats加载到文件中。也可以用print_stats([stream])方法打印结果。它会把结果打印到sys.stdout里，或者任何其他设置成参数的数据流中。

下面的例子和前面的函数一样。这次函数通过line_profiler的API进行性能分析：

```
import line_profiler
import sys


def test():
    for i in range(0, 10):
        print i**2
    print "End of the function"


prof = line_profiler.LineProfiler(test)  # 把函数传递到性能分析器中

prof.enable()  # 开始性能分析
test()
prof.disable()  # 停止性能分析

prof.print_stats(sys.stdout)  # 打印性能分析结果
```

2.3.1 kernprof

kernprof工具和line_profiler是集成在一起的，允许我们从源代码中抽象大多数性能分析代码。这就表示我们可以用它分析应用的性能，和前面做的一样。kernprof将为我们做以下事情。

- 它将和cProfile、lsprof甚至profile模块一起工作，具体要看哪一个性能分析器可用。
- 它会自动寻找脚本文件，如果文件不在当前文件夹，它会检测PATH路径。
- 将实例化分析器，并把名字添加到__builtins__命名空间的profile中。这样我们就可以在代码中使用性能分析器了。在line_profiler示例中，我们甚至可以直接把它当作装饰器用，不需要导入。
- stats性能分析文件可以用pstats.Stats类进行查看，或者使用下面的代码查看。

  ```
  $ python -m pstats stats_file.py.prof
  ```

  或者在lprof文件中查看：

  ```
  $ python -m line_profiler stats_file.py.lprof
  ```

2.3.2 kernprof 注意事项

在读取kernprof的输出结果时，有两件事情需要注意。有时，输出结果可能会比较混乱，或者数字可能没增加到总时间。这些最常见问题的解决方案如下。

- **在性能分析函数调用另一个函数时，没有把每一行消耗的时间增加到总时间上**：当完成一个函数的性能分析时，可能会发生之前的函数分析结果没有加到总时间上的情况。这是因为kernprof只记录函数内部消耗的时间，以免对程序造成额外的负担，如下图所示。

```
Timer unit: 1e-06 s

Total time  0.010539 s
File: kernprof-test3.py
Function: printI at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def printI(i):
     5        10            3      0.3      0.0      counter = 0
     6     20010         5029      0.3     47.7      for a in range(0, 2000):
     7     20000         5427      0.3     51.5          counter +=1
     8        10           80      8.0      0.8      print i ** 2

Total time: 0.019611 s
File: kernprof-test3.py
Function: test at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           @profile
    11                                           def test():
    12        11           39      3.5      0.2      for i in range(0, 10):
    13        10        19567   1956.7     99.8          printI(i)
    14         1            5      5.0      0.0      print "End of the function"
```

之前的例子中显示的情况是：printI函数在性能分析器里消耗了0.010539秒。但是，在test函数内，时间消耗量是19 567个单位时间，共计0.019567秒。

- **分析报告中，列表综合（list comprehension）表达式的Hit比它们实际消耗的要多很多**：基本上是因为对表达式进行性能分析时，分析报告对每次迭代增加了一个Hit。如下图所示。

```
Timer unit: 1e-06 s

Total time: 6.7e-05 s
File: kernprof-test3.py
Function: printExpression at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def printExpression():
     4       102           27      0.3     40.3      myList = [x for x in xrange(0, 50)]
     5         2           40     20.0     59.7      print myList

Total time: 0.00011 s
File: kernprof-test3.py
Function: test at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def test():
     9         1           69     69.0     62.7      printExpression()
    10         1           37     37.0     33.6      printExpression()
    11         1            4      4.0      3.6      print "End of the function"
```

你会看到表达式实际的Hit数是102，printExpression函数每次被调用时需要2次Hit。其他100次Hit是xrange函数消耗的。

2.3.3 性能分析示例

我们已经学习了line_profiler和kernprof的基础知识，下面让我们看一些有趣的例子。

1. 回到斐波那契数列

让我们继续对斐波那契数列进行性能分析。通过对两种性能分析器结果进行比较，我们可以更好地了解两种工作方式。

让我们先看看新的性能分析器的输出结果：

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
Wrote profile results to basic-fibo.py.lprof
Timer unit: 1e-06 s

Total time: 0.039405 s
File: basic-fibo.py
Function: fib at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def fib(n):
     4     57291        12857      0.2     32.6      if n <= 1:
     5     28656         5604      0.2     14.2          return n
     6                                               else:
     7     28635        20944      0.7     53.2          return fib(n-1) + fib(n-2)

Total time: 0.111788 s
File: basic-fibo.py
Function: fib_seq at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           def fib_seq(n):
    11        21            7      0.3      0.0      seq = [ ]
    12        21            6      0.3      0.0      if n > 0:
    13        20           79      4.0      0.1          seq.extend(fib_seq(n-1))
    14        21       111690   5318.6     99.9      seq.append(fib(n))
    15        21            6      0.3      0.0      return seq
```

通过报告中的所有数据，我们可以看出时间并不是问题。在fib函数里，没有一行代码消耗了太多时间（也不应该消耗很多时间）。在fib_seq里面，只有一行消耗了大量时间，但那是因为递归是在fib里面运行的。

所以，我们的问题（其实我们也已经知道）就是递归，以及执行fib函数的次数（共有57 291次）。每次调用函数时，解释器都要按名称查询一次，然后再执行函数。每次调用fib函数时，都需要调用两次。

首先要解决的问题就是降低递归的次数。

我们可以像之前那样重写一个快速的递归函数，或者用装饰器缓存结果。运行结果如下图所示。

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
Wrote profile results to basic-fibo.py.lprof
Timer unit: 1e-06 s

Total time: 4.7e-05 s
File: basic-fibo.py
Function: fib at line 15

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    15                                           @cached
    16                                           @profile
    17                                           def fib(n):
    18        21            7      0.3     14.9      if n <= 1:
    19         2            1      0.5      2.1          return n
    20                                               else:
    21        19           39      2.1     83.0          return fib(n-1) + fib(n-2)

Total time: 0.000225 s
File: basic-fibo.py
Function: fib_seq at line 23

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    23                                           @profile
    24                                           def fib_seq(n):
    25        21            2      0.1      0.9      seq = [ ]
    26        21            5      0.2      2.2      if n > 0:
    27        20           64      3.2     28.4          seq.extend(fib_seq(n-1))
    28        21          149      7.1     66.2      seq.append(fib(n))
    29        21            5      0.2      2.2      return seq
```

Hit数量从57 291将到了21。这又一次证明了装饰器缓存在这个例子中是一个很好的优化方案。

2. 倒排索引

我们不重复使用之前的示例来演示新的性能分析器，而是来看另一个示例：创建倒排索引（http://en.wikipedia.org/wiki/inverted_index）。

倒排索引是许多搜索引擎用来同时在若干文件中搜索文字的工具。它的工作方式是预扫描文件，把内容分割成单词，然后保存单词与文件之间的对应关系（有时也记录单词的位置）。通过这种方式搜索单词时，可以实现$O(1)$时间复杂度（恒定时间）。

让我们看看下面的例子：

```
// 用下面这些文件：
file1.txt = "This is a file"
file2.txt = "This is another file"
// 获得如下索引：
This, (file1.txt, 0), (file2.txt, 0)
is, (file1.txt, 5), (file2.txt, 5)
a, (file1.txt, 8)
another, (file2.txt, 8)
file, (file1.txt, 10), (file2.txt, 16)
```

现在，如果我们要查找单词`is`，我们知道它是在两个文件中（不同的位置）。让我们看看下面计算索引位置的代码（和之前一样，下面的代码中有一些明显需要改进的地方，请你耐心看完，后面会不断优化）。

```
#!/usr/bin/env python

import sys
```

```
import os
import glob


def getFileNames(folder):
    return glob.glob("%s/*.txt" % folder)


def getOffsetUpToWord(words, index):
    if not index:
        return 0
    subList = words[0:index]
    length = sum(len(w) for w in subList)
    return length + index + 1


def getWords(content, filename, wordIndexDict):
    STRIP_CHARS = ",.\t\n |"
    currentOffset = 0

    for line in content:
        line = line.strip(STRIP_CHARS)
        localWords = line.split()
        for (idx, word) in enumerate(localWords):
            word = word.strip(STRIP_CHARS)
            if word not in wordIndexDict:
                wordIndexDict[word] = []
            line_offset = getOffsetUpToWord(localWords, idx)
            index = (line_offset) + currentOffset
            currentOffset = index
            wordIndexDict[word].append([filename, index])
    return wordIndexDict


def readFileContent(filepath):
    f = open(filepath, 'r')
    return f.read().split(' ')


def list2dict(list):
    res = {}
    for item in list:
        if item[0] not in res:
            res[item[0]] = []
        res[item[0]].append(item[1])
    return res


def saveIndex(index):
    lines = []
    for word in index:
        indexLine = ""
        glue = ""
        for filename in index[word]:
```

```
            indexLine += "%s(%s, %s)" % (glue, filename, ','.join(map(str,
index[word][filename])))
            glue = ","
        lines.append("%s, %s" % (word, indexLine))

    f = open("index-file.txt", "w")
    f.write("\n".join(lines))
    f.close()


def __start__():
    files = getFileNames('./files')
    words = {}
    for f in files:
        content = readFileContent(f)
        words = getWords(content, f, words)
    for word in (words):
        words[word] = list2dict(words[word])
    saveIndex(words)


__start__()
```

前面的代码很简单。程序从.txt文件获取任务，那正是我们需要的。它会加载所有的.txt文件，然后分割成单词，计算这些单词在文件中的偏移量，再把这些信息都保存到index-file.txt文件里。

下面我们开始性能分析，看看结果如何。由于我们不知道哪个函数任务繁重，哪个函数任务简单，因此我们给每个函数都加上`@profile`来分析函数性能。

(1) `getOffsetUpToWord`

`getOffsetUpToWord`函数看着像是进行性能优化的合适对象，因为它在执行过程中消耗了比较多的时间。让我们把装饰器加上看看它的性能。

```
Total time: 1.45378 s
File: mapper.py
Function: getOffsetUpToWord at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           @profile
    13                                           def getOffsetUpToWord(words, index):
    14    313868        81878      0.3      5.6    if not index:
    15     29682         7582      0.3      0.5      return 0
    16    284186       106398      0.4      7.3    subList = words[0:index]
    17    284186        70307      0.2      4.8    length = 0
    18   1998159       597693      0.3     41.1    for w in subList:
    19   1713973       514410      0.3     35.4      length += len(w)
    20    284186        75512      0.3      5.2    return length + index + 1
```

(2) `getWords`

`getWords`函数做了大量的动作。它里面有两层`for`循环，所以我们也要在上面使用装饰器。

```
Total time: 4.00185 s
File: mapper.py
Function: getWords at line 23

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    23                                           @profile
    24                                           def getWords(content, filename, wordIndexDict):
    25
    26         2            2      1.0      0.0    STRIP_CHARS = ",.\t\n |"
    27         2            1      0.5      0.0    currentOffset = 0
    28
    29
    30     38858        13798      0.4      0.3    for line in content:
    31     38856        18337      0.5      0.5      line = line.strip(STRIP_CHARS)
    32     38856        35749      0.9      0.9      localWords = line.split()
    33    352724       143714      0.4      3.6      for (idx, word) in enumerate(localWords):
    34    313868       139410      0.4      3.5        word = word.strip(STRIP_CHARS)
    35    313868       170350      0.5      4.3        if word not in wordIndexDict:
    36     42527        19517      0.5      0.5          wordIndexDict[word] = []
    37
    38    313868      3058664      9.7     76.4        line_offset = getOffsetUpToWord(localWords, idx)
    39    313868       113722      0.4      2.8        index = (line_offset) + currentOffset
    40    313868       109872      0.4      2.7        currentOffset = index
    41    313868       178715      0.6      4.5        wordIndexDict[word].append([filename, index])
    42
    43         2            0      0.0      0.0    return wordIndexDict
```

(3) `list2dict`

`list2dict`函数把每个单元是两个元素的数组构成的列表转换成字典。字典把每个数组的第一个元素作为键，第二个元素作为值。我们同样加上`@profile`分析性能。

```
Total time: 0.448933 s
File: mapper.py
Function: list2dict at line 50

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    50                                           @profile
    51                                           def list2dict(list):
    52     42527        14268      0.3      3.2    res = {}
    53    356395       116712      0.3     26.0    for item in list:
    54    313868       139029      0.4     31.0      if item[0] not in res:
    55     46535        14948      0.3      3.3        res[item[0]] = []
    56    313868       154092      0.5     34.3      res[item[0]].append(item[1])
    57     42527         9884      0.2      2.2    return res
```

(4) `readFileContent`

`readFileContent`函数只有两行，就是简单地使用`split`方法对文件内容进行处理。这里没有需要优化的地方，所以我们忽略它，把注意力集中到其他函数上。

```
Total time: 0.003255 s
File: mapper.py
Function: readFileContent at line 45

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    45                                           @profile
    46                                           def readFileContent(filepath):
    47         2           23     11.5      0.7      f = open(filepath, 'r')
    48         2         3232   1616.0     99.3      return f.read().split( '\n' )
```

(5) saveIndex

saveIndex用一种简单的格式生成文件处理的结果。从下面的性能分析结果可以看出，我们可以获得更好的结果。

```
Total time: 0.23337 s
File: mapper.py
Function: saveIndex at line 59

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    59                                           @profile
    60                                           def saveIndex(index):
    61         1            0      0.0      0.0    lines = []
    62     42528        17454      0.4      7.5    for word in index:
    63     42527        14588      0.3      6.3      indexLine = ""
    64     42527        13140      0.3      5.6      glue = ""
    65     89062        37775      0.4     16.2      for filename in index[word]:
    66     46535       105780      2.3     45.3        indexLine += "%s(%s, %s)" % (glue, filename, ','.join(map(str, index[word][filename])))
    67     46535        15861      0.3      6.8        glue = ","
    68     42527        24191      0.6     10.4      lines.append("%s, %s" % (word, indexLine))
    69
    70         1          423    423.0      0.2    f = open("index-file.txt", "w")
    71         1         3783   3783.0      1.6    f.write("\n".join(lines))
    72         1          375    375.0      0.2    f.close()
```

(6) __start__

最后是主方法__start__，它主要就是调用其他函数，没有什么性能负担，所以我们同样忽略它。

```
Total time: 6.09869 s
File: mapper.py
Function: __start__ at line 74

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    74                                           @profile
    75                                           def __start__():
    76         1          387    387.0      0.0    files = getFileNames('./files')
    77         1            0      0.0      0.0    words = {}
    78         3            0      0.0      0.0    for f in files:
    79         2         3400   1700.0      0.1      content = readFileContent(f)
    80         2      4923513 2461756.5     80.7      words = getWords(content, f, words)
    81     42528        16516      0.4      0.3    for word in (words):
    82     42527       796925     18.7     13.1      words[word] = list2dict(words[word])
    83         1       357948 357948.0      5.9    saveIndex(words)
```

综上所述，我们之前分析了6个函数的性能，忽略了其中两个函数，因为它们要么太简单，要么没有值得关心的内容。于是我们一共有4个函数需要优化。

(1) getOffsetUpToWord

让我们看看第一个函数getOffsetUpToWord，里面许多行代码就是简单地把单词的长度增加到当前的索引位置。有一种更加具有Python风格的方式，让我们试一试。

原函数运行共消耗了1.4秒，让我们简化代码来缩短程序运行时间。增加单词长度的代码可以缩短，如下所示：

```
def getOffsetUpToWord(words, index):
    if(index == 0):
        return 0
    length = reduce(lambda curr, w: len(w) + curr, words[0:index], 0)
    return length + index + 1
```

代码简化只是把多余的变量声明和查询取消了。这好像没什么。但是，如果我们运行代码，时间会降到0.9秒。不过代码里面还是有一个明显的缺陷，就是lambda表达式。每当我们调用

getOffsetUpToWord函数时，都要动态地创建一个函数。我们一共调用了313 868次，所以更好的办法是事先创建好函数。我们在reduce表达式里面使用函数引用就可以了，如下所示：

```
def addWordLength(curr, w):
    return len(w) + curr

@profile
def getOffsetUpToWord(words, index):
    if(index == 0):
        return 0
    length = reduce(addWordLength, words[0:index], 0)
    return length + index + 1
```

输出结果如下图所示。

```
Total time: 0.816949 s
File: mapper.py
Function: getOffsetUpToWord at line 13

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    13                                           @profile
    14                                           def getOffsetUpToWord(words, index):
    15    313868        80314      0.3      9.8    if not index:
    16     29682         7050      0.2      0.9      return 0
    17    284186       652906      2.3     79.9    length = reduce(addWordLength, words[0:index], 0)
    18    284186        76679      0.3      9.4    return length + index + 1
```

通过一点小改进，执行时间降到了0.8秒。在上面的截图中，我们还发现函数的前两行仍然消耗了大量不想要的Hit（也是时间）。if检测语句没必要，因为reduce表达式的初始值就是0。长度变量声明没有必要，我们可以直接返回长度、索引和整数1的和。

按照这个思路修改代码，如下所示：

```
def addWordLength(curr, w):
    return len(w) + curr

@profile
def getOffsetUpToWord(words, index):
    return reduce(addWordLength, words[0:index], 0)  + index + 1
```

这样函数的总运行时间就从1.4秒降到了0.67秒。

(2) getWords

让我们来看下一个函数：getWords。这个函数非常慢，从前面的截屏可以看出，它的运行时间长达4秒。这实在很糟糕，让我们看看是怎么回事。首先，函数中最费时的代码行是调用getOffsetUpToWord函数。由于我们前面已经优化过getOffsetUpToWord函数，所以现在运行时间从原来的4秒降低到了2.2秒。

这里对副作用的优化非常合理，但是我们还可以进一步优化。我们用了一个wordIndexDict词典变量，所以在插入新键之前需要先检查键存不存在。在函数中做这个检查要消耗大约0.2秒时间。虽然耗时不多，但仍然可以优化。要消除检查，我们可以用defaultdict类。它是dict

的子类，只是增加了一个功能。如果键不存在，就使用预先设置的默认值。这样就可以为程序运行节省0.2秒。

另一个实用的小优化是变量的声明。虽然看着是小事，但是调用了313 868次就无疑要消耗一些时间了。因此，让我们看看这几行性能分析结果：

```
35    313868    1266039    4.0    62.9    line_offset = getOffsetUpToWord(localWords,
       idx)
36    313868     108729     0.3     5.4    index = (line_offset) + currentOffset
37    313868     101932     0.3     5.1    currentOffset = index
```

这三行代码可以用一行代码搞定，如下所示：

```
currentOffset += getOffsetUpToWord(localWords, idx)
```

这样我们就又缩减了0.2秒。最后我们对每一行和每个单词都进行了`strip`操作。我们可以在加载文件的时候，对文件内容使用几次`replace`方法来进行简化。这样既将要处理的文本清理干净了，又消除了在`getWords`函数里查询和调用方法的时间。

新的代码如下：

```
def getWords(content, filename, wordIndexDict):
    currentOffset = 0
    for line in content:
        localWords = line.split()
        for (idx, word) in enumerate(localWords):
            currentOffset += getOffsetUpToWord(localWords, idx)
            wordIndexDict[word].append([filename, currentOffset])]])
    return wordIndexDict
```

现在只需要1.57秒了。还有一个优化值得我们看看。这个优化适合我们的例子，因为`getOffsetUpToWord`函数只用了一次。由于这个函数只有一行，我们可以把这一行直接写入`getWords`。这样可以把时间减少到1.07秒（减少了0.5秒）。下面就是最新版函数的样子：

```
Total time: 1.07077 s
File: mapper.py
Function: getWords at line 21

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    21                                           @profile
    22                                           def getWords(content, filename, wordIndexDict):
    23         2            3      1.5      0.0    currentOffset = 0
    24     38858        11974      0.3      1.1    for line in content:
    25     38856        34947      0.9      3.3      localWords = line.split()
    26    352724       120798      0.3     11.3      for (idx, word) in enumerate(localWords):
    27    313868       709905      2.3     66.3        currentOffset = reduce(addWordLength, localWords[0:idx], 0)  + idx + 1 + currentOffset
    28    313868       193142      0.6     18.0        wordIndexDict[word].append([filename, currentOffset])
    29         2            0      0.0      0.0    return wordIndexDict
```

如果你还要在其他地方调用这个函数，这么做不方便维护代码。开发过程中代码的可维护性也是非常重要的一个方面。当你要确定何时停止优化时，代码的可维护性可以作为一个重要的决定因素。

(3) `list2dict`

对于`list2dict`函数没有什么可以优化的，不过我们可以让它变得更易读，而且可以减少约

0.1秒的时间。我们又一次为了代码可读性而放弃对时间的执着。我们可以再一次使用defaultdict类，去掉检查环节。最终代码如下：

```
def list2dict(list):
    res = defaultdict(lambda: [])
    for item in list:
        res[item[0]].append(item[1])
    return res
```

这样处理后，代码行数更少，更方便阅读，也更容易理解。

(4) saveIndex

最后，让我们看看saveIndex函数。通过之前的分析报告，可以看到一共用了0.23秒完成索引文件的预处理和保存。这个性能已经很好了，不过我们还可以对字符串连接进行一点优化。

保存数据之前，我们把一些字符串组合起来构成一个单词。在同样的循环体中，我们还重置了indexLine和glue变量。这些操作放在一起消耗了大量的时间，所以我们应该改变策略。

优化后的代码如下：

```
def saveIndex(index):
    lines = []
    for word in index:
        indexLines = []
        for filename in index[word]:
            indexLines.append("(%s, %s)" % (filename,
','.join(index[word][filename])))
        lines.append(word + "," + ','.join(indexLines))
    f = open("index-file.txt", "w")
    f.write("\n".join(lines))
    f.close()
```

你会看到，在前面的代码中，我们改变了for循环结构。现在不是把新的字符串加入indexLine变量，而是追加到列表里。我们还去掉了map调用，这样直接调用join就可以处理字符串。map函数被移动到了list2dict函数内，在添加字符串到列表时，直接用索引即可。

最后我们用+操作符连接字符串，而不是用C语言字符串的连接方式（%），后者耗时更多。最终，函数的执行时间从0.23降到了0.13秒，速度提升了0.1秒。

2.4 小结

这一章介绍了两个Python性能分析器：cProfile，是语言自带的；line_profiler，可以让我们看到每一行代码的性能。我们还介绍了一些使用它们分析和优化代码的示例。

在下一章，我们将看到一些可视化工具，在工作中可以帮助我们展示本章出现的性能分析数据，但它们是通过图形的方式展示数据的。

第 3 章

可视化——利用GUI理解性能分析数据

3

虽然前面两章已经介绍了性能分析方法，但是整个分析过程中我们仿佛置身于黯淡无光的黑夜之中[1]。我们一直都在观察各种性能分析数据。我们试图通过努力，不断地降低Hit次数、运行时间，以及优化其他性能指标。但是这些数字表达的实际意义有时难以理解。

根据眼前的性能分析结果，我们很难观察整个程序的全貌。如果系统再复杂一点儿，要从全局观察性能分析结果就会更加困难。

由于我们是人而非计算机，所以当拥有可视化的辅助手段时，我们的工作效果会更好。在性能分析过程中，如果能够更好地理解数据的意义将大有裨益。为此，我们需要使用可视化工具来展示上一章见过的数据。这些工具将会给予我们许多帮助。通过它们可以快速定位问题，解决性能瓶颈。另外，我们也会对系统有更全面的认识。

这一章将介绍两种可视化工具。

- KCacheGrind/`pyprof2calltree`：这套组合工具可以把`cProfile`的输出结果转换成KCacheGrind支持的格式，从而帮助我们实现数据可视化。
- RunSnakeRun（http://www.vrplumber.com/programming/runsnakerun/）：这个工具也可以把`cProfile`的输出结果可视化。它还带有方块图和可排序的列表。

对于每个工具，我们都会介绍基本的安装方法和用户界面。然后，我们将用这些工具对第2章中示例的性能输出结果进行分析。

3.1 KCacheGrind/`pyprof2calltree`

我们要看的第一个GUI工具是KCacheGrind。这个数据可视化工具可以用来分析和展示多种格式的性能分析数据。在示例中，我们将使用`cProfile`的输出数据。要实现数据可视化，我们

① 作者是在说命令行工具。——译者注

还需要使用命令行工具pyprof2calltree。

这个工具是著名的lsprofcalltree.py（https://people.gnome.org/~johan/lsprofcalltree.py）项目的升级版。它更像是Debian系统里的kcachegrind-converter（https://packages.debian.org/en/stable/kcachegrind-converters）。我们将通过这个工具把cProfile的输出结果转换成KCacheGrind可以读取的形式。

3.1.1 安装

安装pyprof2calltree之前，需要先安装Python包管理器pip。然后使用下面的命令安装pyprof2calltree即可：

```
$ pip install pyprof2calltree
```

需要注意的是，这个安装步骤和安装指令都默认是在Ubuntu 14.04 Linux发行版上运行的，除非有其他提示。

现在，再安装KCacheGrind，其安装方法有点不同。它是KDE桌面环境的一部分，所以如果你已经安装了这个桌面环境，那么KCacheGrind默认已经安装好了。但是，如果你没有KDE环境（假如你用的是Gnome），那么你可以通过操作系统的包管理器来安装。例如在Ubuntu系统中，安装命令如下所示：

```
$ sudo apt-get install kcachegrind
```

> 这个命令运行之后，可能需要安装一些与工具没有直接关系，但是与KDE环境有依赖的软件包。具体安装的速度就由你的网速决定了。

Windows和OS X用户可以安装KCacheGrind的分支版本QCacheGrind，它是一个已编译过的可执行文件。

Windows用户可以从http://sourceforge.net/projects/qcachegrindwin/下载，OS X用户可以通过下面的命令安装：

```
$ brew install qcachegrind
```

3.1.2 用法

pyprof2calltree模块有两种用法：一种是通过命令行加参数的形式，另一种是在REPL（read-eval-print loop，读取–求值–输出循环）交互式编程环境里运行（也可以根据需要在性能分析的脚本中运行）。

第一种用法（命令行形式）在我们已经有性能分析输出文件时，使用很方便。使用这个工具，通过下面的命令就可以获得结果：

```
$ pyprof2calltree -o [output-file-name] -i input-file.prof
```

有一些参数可以帮助我们处理不同的情况。其中两个参数解释如下。

- -k：如果想立即运行KCacheGrind，就可以加上这个参数。
- -r：如果还没有性能分析数据，可以用这个参数直接分析Python脚本文件生成最终结果。

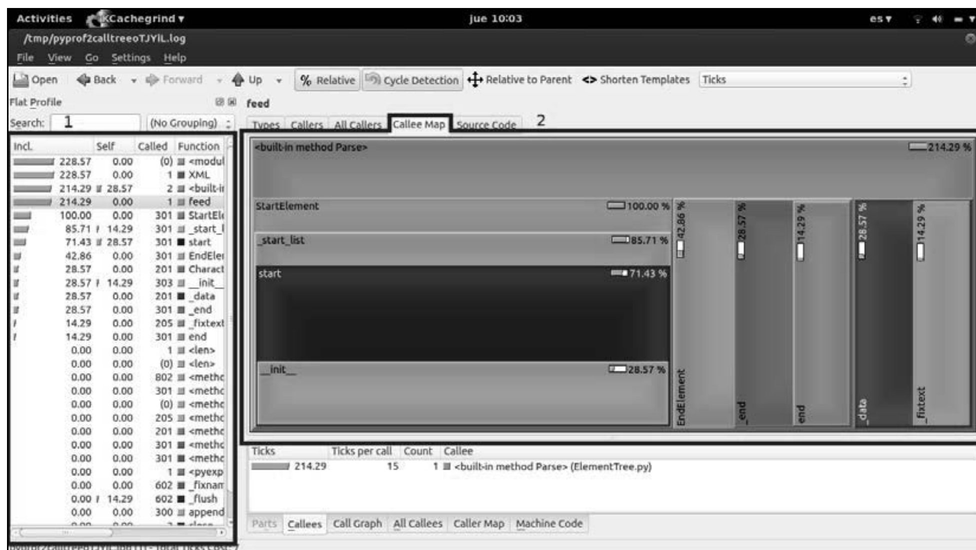
如果想在REPL里面使用它，可以从pyprof2calltree包里导入convert或visualize函数（也可以两个都导入）。第一个函数会输出性能分析结果文件，第二个函数会直接启动KCacheGrind显示结果。

示例如下：

```
from xml.etree import ElementTree
from cProfile import Profile
import pstats
xml_content = '<a>\n' + '\t<b/><c><d>text</d></c>\n' * 100 + '</a>'
profiler = Profile()
profiler.runctx(
    "ElementTree.fromstring(xml_content)", locals(), globals())

from pyprof2calltree import convert, visualize
stats = pstats.Stats(profiler)
visualize(stats)  # 运行kcachegrind
```

代码将直接运行KCacheGrind，结果如下面的截图所示。

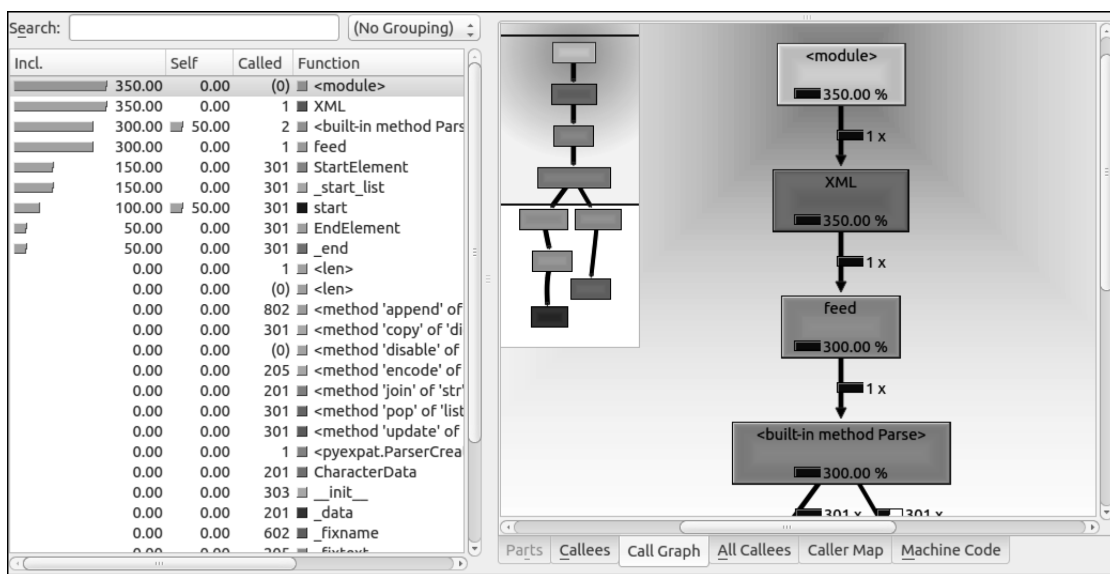


在上面的截图中，我们可以看到左边(1)是我们在前一章看到的结果，在右边(2)我们选了一个标签——Callee Map。Callee Map里面显示了一些矩形，每个矩形与左边列表里对应函数的消耗时间一致。

在左边窗口中，有两列需要注意。

- ❑ Incl.列：这个指标表示函数的累计消耗时间。就是说它会把函数消耗的时间和其他被它调用的函数消耗的时间加起来统计。如果在这一列中函数消耗的时间很高，并不一定是这个函数消耗的时间很长，也可能是它调用的函数运行时间很长。
- ❑ Self列：只包含函数本身消耗的时间，不包括它调用的函数需要的时间。因此，如果函数的Self值很高，表示这个函数本身消耗的时间很长，它可能是性能优化的起点。

另一个有用的可视化图是函数调用关系图（Call Graph），选择一个函数后可以在右下角选项卡里打开函数调用图，它将显示函数调用的具体过程（调用的次数）。上面示例的结果如下图所示。



3.1.3 性能分析器示例：TweetStats

现在让我们回到第2章的示例，看看如何用`pyprof2calltree/kcachegrind`组合套件分析性能。

这次我们不用斐波那契数列的例子了，因为那个例子非常简单，而且我们也已经做过两遍了。所以我们这里使用TweetStats示例。程序会直接读取一组推文，然后进行一些统计。我们没有调

整代码，所以直接参考第2章的示例即可。

由于原来的脚本性能分析方式是打印性能统计信息，所以我们要做一些调整。你会看到程序只做了一点小改动：

```
import cProfile
import pstats
import sys

from tweetStats import build_twit_stats

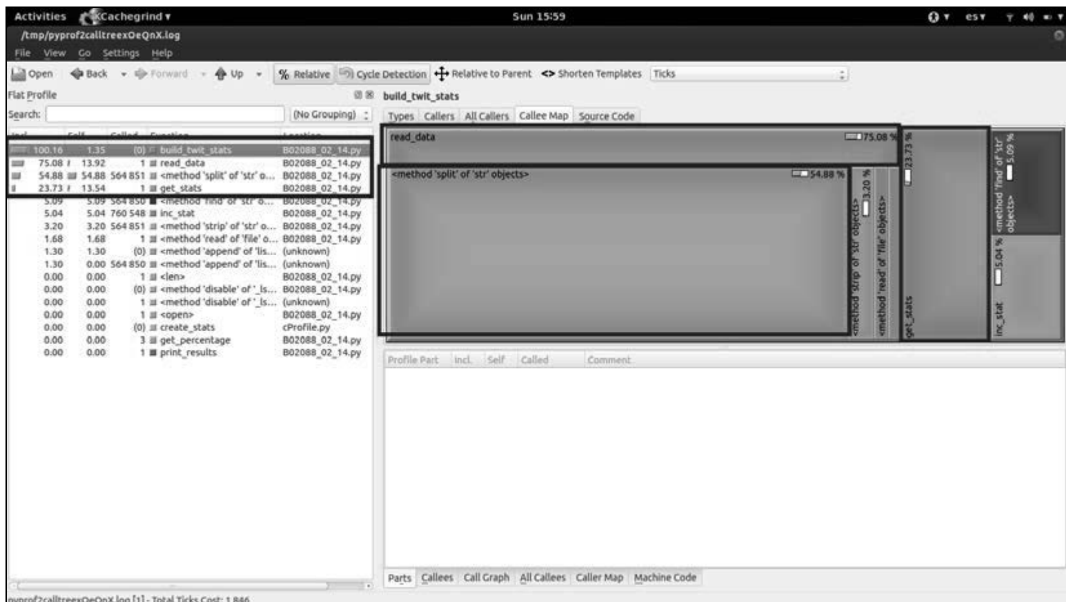
profiler = cProfile.Profile()
profiler.enable()


build_twit_stats()
profiler.create_stats()
stats = pstats.Stats(profiler)
# 把stats保存到tweet-stats.prof文件里，
# 而不是打印到命令行中。
stats.strip_dirs().sort_stats('cumulative').dump_stats('tweet-stats.prof')
```

程序运行后，性能统计信息保存在tweet-stats.prof文件中。我们可以用下面的命令把文件转换成可视化图形：

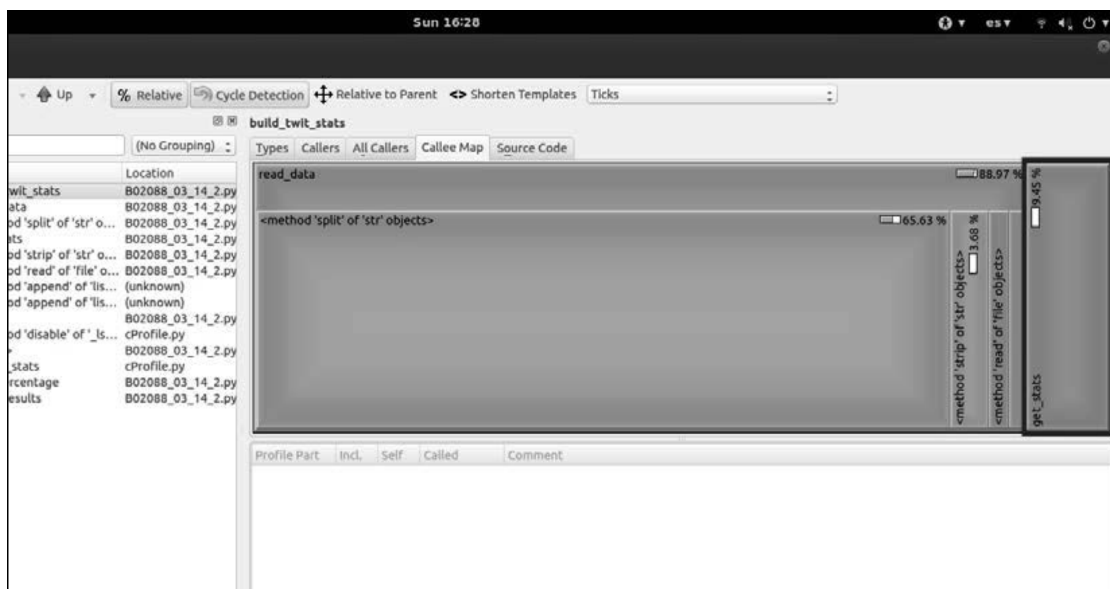
```
$pyprof2calltree -i tweet-stats.prof -k
```

对应的可视化效果如下面的截图所示。



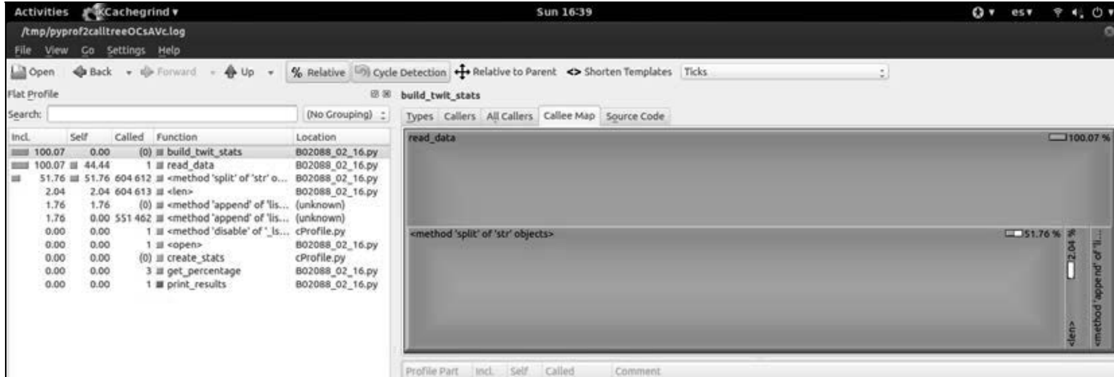
和之前一样，通过`build_twit_stats`函数的Callee Map，我们可以看到整个程序的函数调用情况。里面的瓶颈很明显（右边最大的几个矩形）：`read_data`、`split`以及右边的方块`get_stats`方法。

在`get_stats`函数矩形里面，我们可以看到函数是如何构成的：`inc_stat`函数和Python字符串的`find`函数。我们知道第一个函数是示例代码里的。这个函数非常短，所以消耗的时间应该都是函数查询累计的（不过我们也调用了大约76万次）。`find`方法也是如此。由于它被我们调用的次数太多，所以总的函数查询时间也很明显。让我们用一个非常简单的方法来改造这个函数。首先把`inc_stat`函数删掉，然后把它的内容内联到`get_stats`函数里，之后再把字符串的`find`方法改成`in`方法。结果将会如下面的截图所示。



Callee Map发生了显著的变化。我们会发现`get_stats`函数不再调用其他函数了，因此查询时间就没有了。现在它只占用9.45%的运行时间，比原来降低了23.73%。

上面的结论和我们在前一章得出的结论一样，只不过这次我们是通过另一种方法得到的。让我们继续按照上一章的优化过程观察函数Callee Map的变化。



在上面的截图中，我们选择左边的`build_twitt_stats`函数（在左边的列表中），会发现里面调用的函数都是字符串的简单方法。

令人遗憾的是，KCacheGrind不能显示程序运行消耗的总时间。但是，Callee Map已经清晰地展示了代码简化和优化的结果。

3.1.4　性能分析器示例：倒排索引

让我们再看看第2章的示例：倒排索引。让我们对代码稍作修改以生成性能统计文件，方便后面用KCacheGrind进行分析。

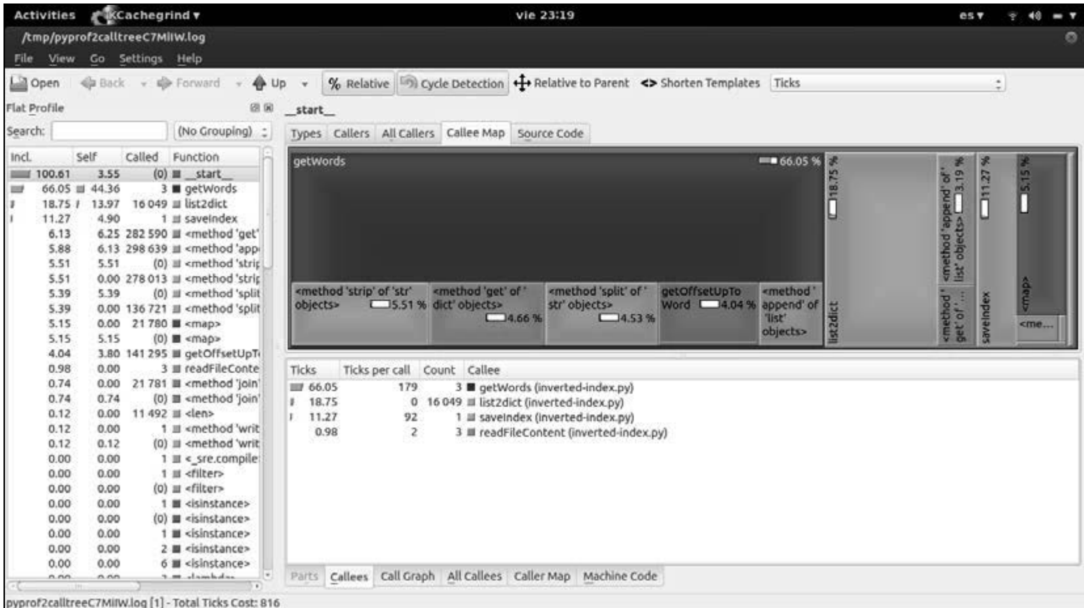
我们需要修改的是代码的最后一行，不需要调用`__start__`函数。代码修改结果如下：

```
profiler = cProfile.Profile()
profiler.enable()
__start__()
profiler.create_stats()
stats = pstats.Stats(profiler)
stats.strip_dirs().sort_stats('cumulative').dump_stats('inverted-index-stats.prof'
)
```

现在运行脚本就会生成一个inverted-index-stats.prof文件。然后，我们可以用下面的命令行启动KCacheGrind：

```
$ pyprof2calltree -i inverted-index-stats.prof -k
```

首先看到的结果如下所示：



首先对左边的列表按照Self字段进行排序，这样我们就可以看到内部消耗时间最长的函数了（并不是函数总共消耗的时间，不包含对其他函数的调用时间）。排序后结果如下所示：

Incl.	Self	Called	Function
66.05	44.36	3	getWords
18.75	13.97	16 049	list2dict
6.13	6.25	282 590	<method 'get'
5.88	6.13	298 639	<method 'app
5.51	5.51	(0)	<method 'strip
5.39	5.39	(0)	<method 'split
5.15	5.15	(0)	<map>
11.27	4.90	1	saveIndex
4.04	3.80	141 295	getOffsetUpT
100.61	3.55	(0)	__start__
0.74	0.74	(0)	<method 'join'
0.12	0.12	(0)	<method 'writ
0.00	0.12	1	<len>
5.51	0.00	278 013	<method 'strip
5.39	0.00	136 721	<method 'split
5.15	0.00	21 780	<map>
0.98	0.00	3	readFileConte
0.74	0.00	21 781	<method 'join'
0.12	0.00	11 492	<len>
0.12	0.00	1	<method 'writ
0.00	0.00	1	<_sre.compile
0.00	0.00	1	<filter>
0.00	0.00	(0)	<filter>

因此，通过前面的列表可以看出，目前最成问题的两个函数是`getWords`和`list2dict`。

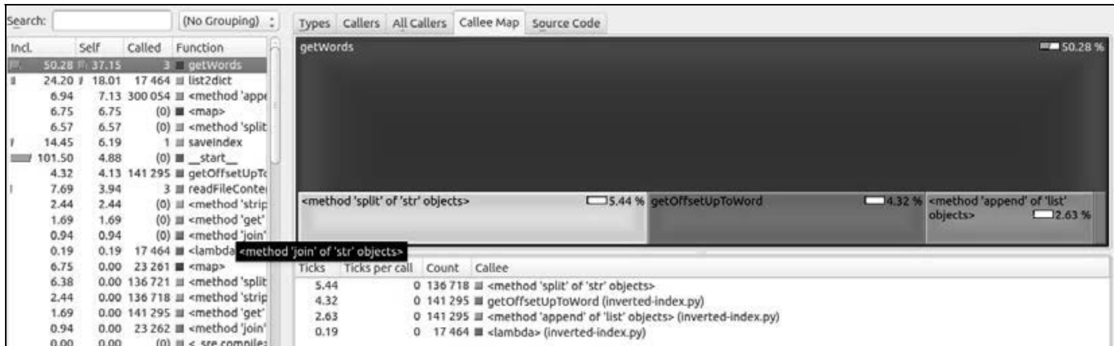
第一个函数可通过以下几种方式改进。

- ❑ wordIndexDict可以改成defaultdict类型，这样做可以把if语句省去。
- ❑ 为了简化代码，readFileContent函数里的strip语句也可以去掉。
- ❑ 许多赋值语句也可以去掉，直接使用最终结果可以节省一些时间。

这样我们的getWords函数如下所示：

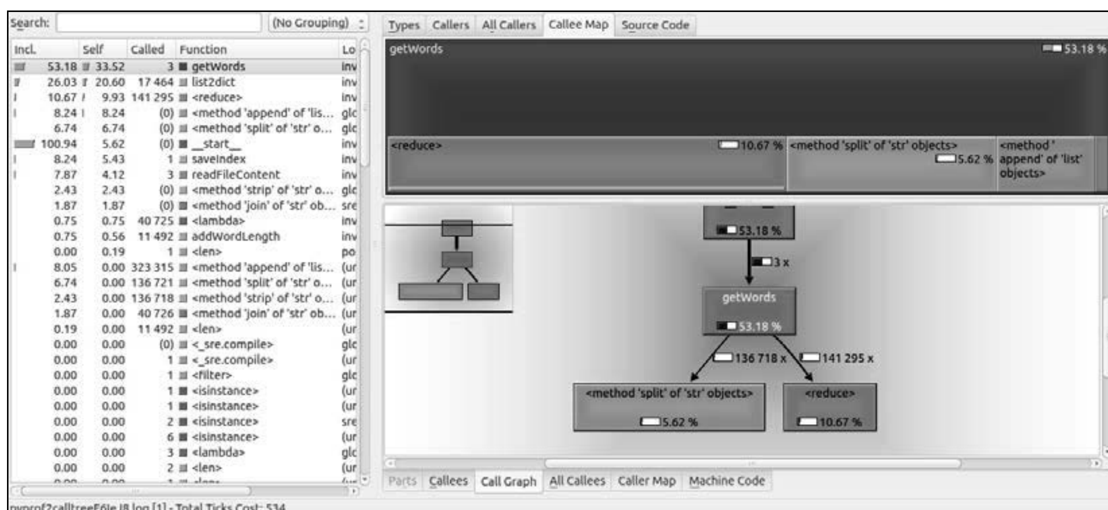
```
def getWords(content, filename, wordIndexDict):
    currentOffset = 0
    for line in content:
        localWords = line.split()
        for (idx, word) in enumerate(localWords):
            currentOffset = getOffsetUpToWord(localWords, idx) +
            currentOffset
            wordIndexDict[word].append([filename, currentOffset])
    return wordIndexDict
```

现在，如果我们再运行KCacheGrind，会发现Callee Map和性能数据会有一点不同。



可以看出，函数的总运行时间（Incl.列）和内部运行时间（Self列）都减少了。但是，在离开这个函数之前还有一个细节值得我们关注。getWords函数一共调用了getOffsetUpToWord函数141 295次，全部消耗在查询时间上面，值得我们好好看看。下面我们就来解决这个问题。

在上一章我们已经解决了这个问题。我们可以把getOffsetUpToWord函数改写成一行，这样就可以把它直接内联到getWords里面，避免查询时间。解决之后让我们再看看性能Callee Map。



现在，总运行时间反而增加了，不过不用担心。这是因为一个函数调用其他函数的时间减少，会导致总时间也发生变化。但是，我们需要关心的函数本身的运行时间（Self列）降低了约4%。

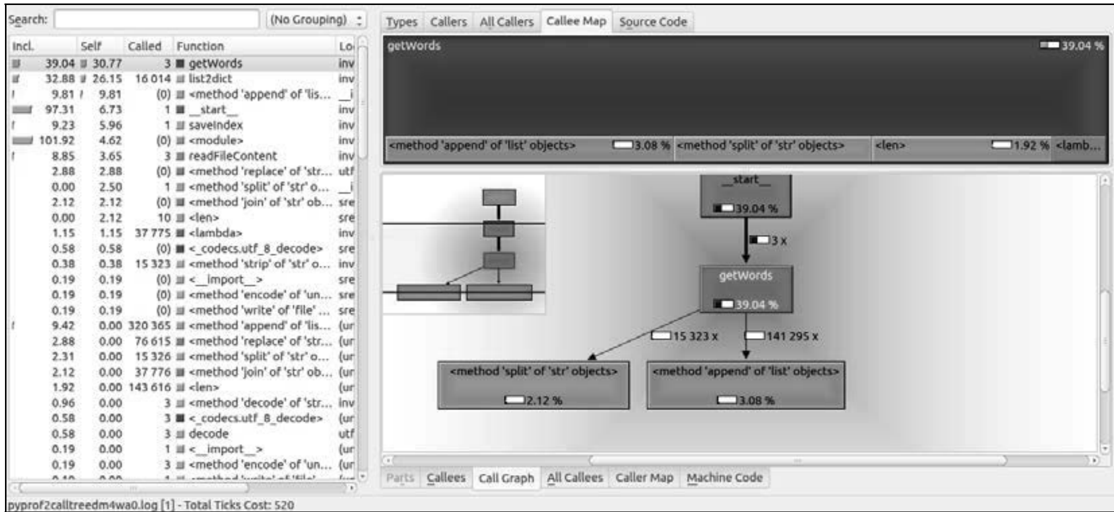
前面的截图还显示了函数调用关系图，从图中可以看出，即使我们做了改进，`reduce`函数依然要调用100 000次以上。如果你仔细看`getWords`函数的代码，会发现我们不需要`reduce`函数。这是因为每次调用时我们都要把前面的调用次数加上，其实我们可以简化代码：

```
def getWords(content, filename, wordIndexDict):
    currentOffset = 0
    prevLineLength = 0
    for lineIndex, line in enumerate(content):
        lastOffsetUptoWord = 0
        localWords = line.split()

        if lineIndex > 0:
            prevLineLength += len(content[lineIndex - 1]) + 1
        for idx, word in enumerate(localWords):
            if idx > 0:
                lastOffsetUptoWord += len(localWords[idx-1])
            currentOffset = lastOffsetUptoWord + idx + 1 + prevLineLength

            wordIndexDict[word].append([filename, currentOffset])
```

运行调整过的代码，会看到分析结果又变化了。



现在函数运行时间（Incl.列）明显降低了，因此函数消耗的时间更少了（正是我们所期望的）。内部运行时间（Self列的值）也下降了，这说明我们用更少的时间完成了同样的事情（因为这列数据的时间不包含调用其他函数的时间）。

3.2　RunSnakeRun

RunSnakeRun是另一个可对性能分析结果进行可视化的工具，可以帮助我们理解数据。这个项目是KCacheGrind的简化版。KCacheGrind也适用于C和C++开发者，而RunSnakeRun是专门为Python开发者定制的。

之前在用KCacheGrind的时候，如果你想看到`cProfile`的图形，需要用其他工具（`pyprof2calltree`）打开。现在不需要了，RunSnakeRun知道如何读取和解释分析结果，所以我们只要设置好文件路径就行了。

这个工具可以提供的特征如下所示。

- 可排序的网格视图，包括：
  - 函数名称
  - 总调用次数
  - 累计时间
  - 文件名和行号
- 函数的具体调用信息，比如函数的调用者和被调用者名称
- 面积与函数运行时间成正比的方块图

3.2.1 安装

安装这个工具之前，首先需要安装一些依赖包，主要有下面三个依赖：

- Python性能分析器
- wxPython（2.8或以上版本，http://www.wxpython.org/）
- Python 2.5或以上版本，暂时不支持Python 3.x版本

执行安装命令还需要用pip（https://pypi.python.org/pypi/pip）。

因此，在安装之前请保证这些依赖都已安装好。如果你用的是基于Debian的Linux发行版（比如Ubuntu），你可以用下面的命令行确保所有的依赖都得到安装（前提是Python已经安装好了）：

```
$ apt-get install python-profiler python-wxgtk2.8 python-setuptools
```

> 对于Windows和OS X用户，需要先安装适合自己系统的依赖包可执行文件。

之后用下面的命令安装即可：

```
$ pip install SquareMap RunSnakeRun
```

然后，就可以使用了。

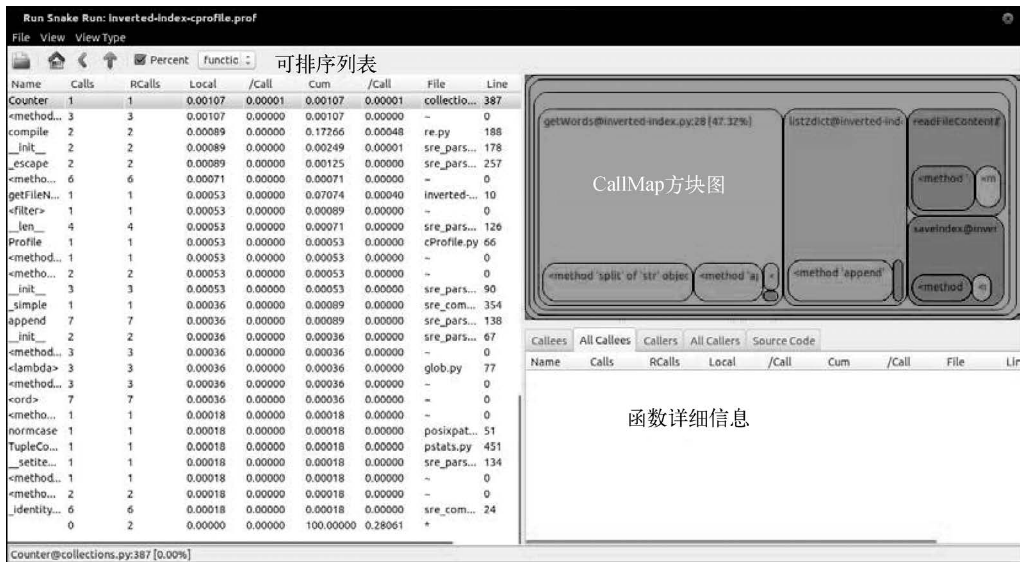
3.2.2 使用方法

现在，为了尽快上手使用，让我们回到前面的例子inverted-index.py。

我们还是用cProfile作为性能分析器把分析结果输出到一个文件里。然后，调用runsnake打开文件：

```
$ python -m cProfile -o inverted-index-cprof.prof inverted-index.py
$ runsnake inverted-index-cprof.prof
```

截图如下所示：



从上面的截图中会发现三个有趣的区域。

- 可排序列表，里面包含了`cProfile`输出的所有数据。
- 函数详细信息区域，里面包含了调用函数（Callers）、被调用函数（Callees）和源代码（Source Code）等标签。
- 方块图区域，用图形显示运行的函数调用关系树。

> 这个GUI工具的优点之一是，当你点击左边列表中的函数时，右边的方块就会高亮显示。如果你点击右边的方块，左边的列表中对应的函数也会高亮显示。

3.2.3　性能分析示例：最小公倍数

让我们用一个简单的小函数来演示这个GUI工具的用法。

我们用的例子是寻找两个正整数的最小公倍数。虽然这是一个非常基础的函数，在网上很容易找到，但是用它来体验这个GUI工具十分恰当。

代码如下所示：

```
def lowest_common_multiplier(arg1, arg2):
    i = max(arg1, arg2)
    while i < (arg1 * arg2):
        if i % min(arg1, arg2) == 0:
            return i
        i += max(arg1, arg2)
    return(arg1 * arg2)
```

```
print lowest_common_multiplier(41391237, 2830338)
```

我相信你看过代码之后已经发现了需要优化的地方，不过请让我用可视化工具来演示。我们首先分析这段代码的性能，然后把结果加载到RunSnakeRun里面。

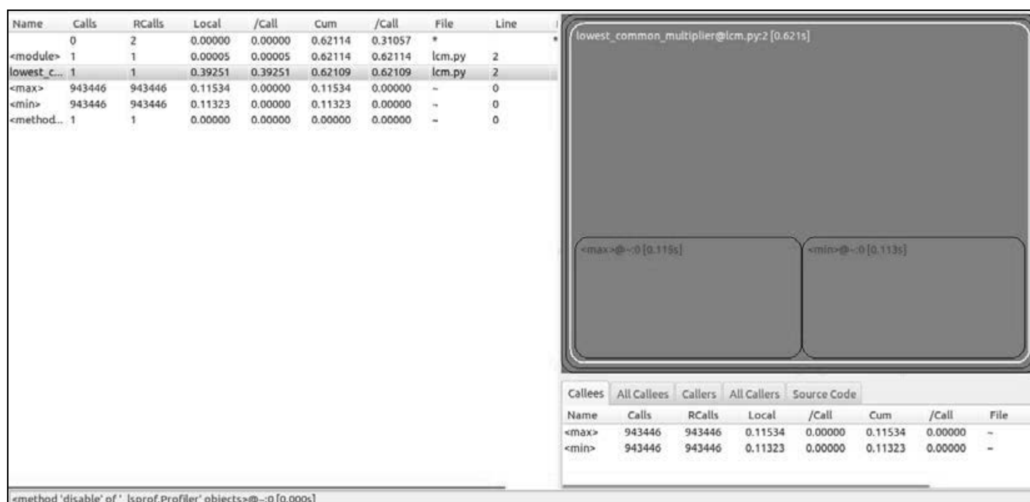
用下面的命令就可以加载：

```
$ python -m cProfile -o lcm.prof lcm.py
```

再用这行命令启动RunSnakeRun：

```
$ runsnake lcm.prof
```


这就是我们的结果：



有一件事情我们之前没有提到，其实是给方块图锦上添花，即在方块的名称旁边可以显示函数运行的具体时间。

因此，观察上图我们会发现一些问题。

- 我们看到在函数运行的总时间——0.621秒里，`max`和`min`一共只消耗了0.228秒。因此，可以认为函数消耗了比简单的`max`和`min`函数更多的时间。
- 我们还发现`max`和`min`调用了943 446次。无论函数查询时间多么短，调用100万次也会是很长的时间。

让我们修改一下代码，再用蛇之眼（eyes of the snake[1]）看看：

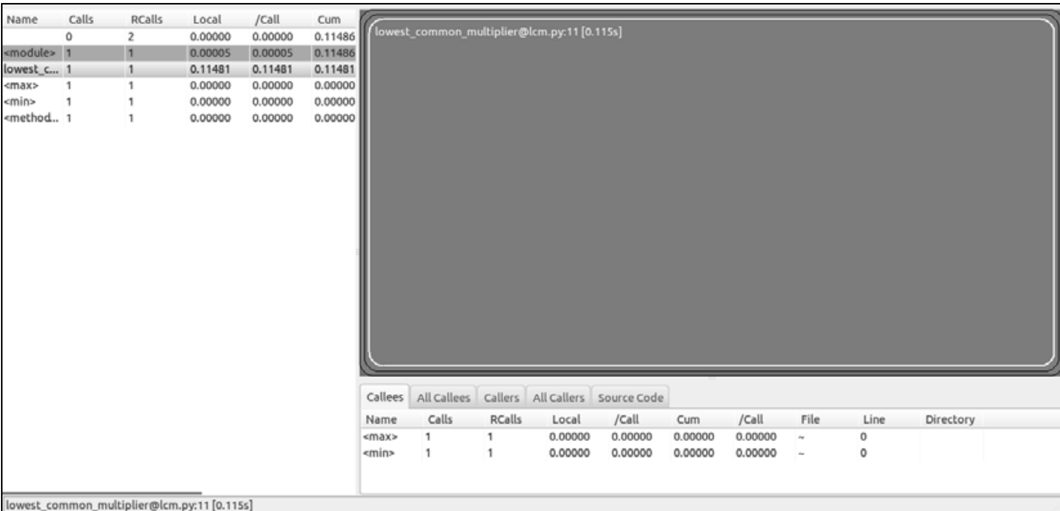
---

① 作者指的是可视化工具RunSnakeRun。——译者注

```
def lowest_common_multiplier(arg1, arg2):
    i = max(arg1, arg2)
    _max = i
    _min = min(arg1,arg2)
    while i < (arg1 * arg2):
        if i % _min == 0:
            return i
        i += _max
    return(arg1 * arg2)

print lowest_common_multiplier(41391237, 2830338)
```

我们将看到类似下面截图的结果：



现在，max和min都没有出现在方块图上，因为我们只调用了它们一次，函数也从0.6秒降到了0.1秒。这就是去掉多余函数查询的效果。

下面再让我们看看另一个更复杂，也更有趣的函数，亟待性能优化。

3.2.4　性能分析示例：用倒排索引查询

在前一章中，我们已经从不同的角度分析了倒排索引的代码。由于我们通过不同的视角并使用不同的方法分析过它，所以代码性能已经很好了。所以，如果我们再用RunSnakeRun分析一遍没什么意义，因为这个工具和之前的工具（KCacheGrind）差不多。

因此，我们将使用倒排索引生成的索引结果，自己编写一个使用该索引结果的搜索脚本。我们将要分析的函数很简单，它只查询索引中的一个单词。具体的方法也很简单。

(1) 把索引结果加载到内存。
(2) 搜索单词并抓取索引信息。
(3) 解析索引信息。
(4) 对每个索引信息，查询相关的文件，然后把包含单词的句子提取出来。
(5) 打印结果。

下面是代码的初始版本：

```
import re
import sys

# 把倒排索引中的项目转换成词典对象，
# 以单词为索引键
def list2dict(l):
    retDict = {}
    for item in l:
        lineParts = item.split(',(')
        word = lineParts.pop[0]
        data = ','.join(lineParts)
        indexDataParts = re.findall('\(([a-zA-Z0-9\./, ]{2,})\)', data)
        retDict[word] = indexDataParts
    return retDict

# 把索引的内容加入内存进行分析
def loadIndex():
    indexFilename = "./index-file.txt"
    with open(indexFilename, 'r') as fh:
        indexLines = []
        for line in fh:
            indexLines.append(line)
        index = list2dict(indexLines)

        return index

# 读取文件内容，单词使用UTF-8编码格式，
# 移除不需要的单词（不希望出现在索引中的单词）
def readFileContent(filepath):
    with open(filepath, 'r') as f:
        return [x.replace(",", "").replace(".", "").replace("\t", "").replace("\r",
"").replace("|", "").strip(" ") for x in
f.read().decode("utf-8-sig").encode("utf-8").split('\n')]

def findMatch(results):
    matches = []
    for r in results:
        parts = r.split(',')
        filepath = parts.pop[0]
        fileContent = ' '.join(readFileContent(filepath))
        for offset in parts:
            ioffset = int(offset)
            if ioffset > 0:
```

```
                ioffset -= 1
            matchLine = fileContent[ioffset:(ioffset + 100)]
            matches.append(matchLine)
    return matches

# 在索引文件中搜索单词
def searchWord(w):
    index = None
    index = loadIndex()
    result = index.get(w)
    if result:
        return findMatch(result)
    else:
        return []

# 让用户自定义单词……
searchKey = sys.argv[1] if len(sys.argv) > 1 else None

if searchKey is None:  # 如果没有搜到，就打印一条信息
    print "Usage: python search.py <search word>"
else:  # 如果单词存在，则进行搜索
    results = searchWord(searchKey)
    if not results:
        print "No results found for '%s'" % (searchKey)
    else:
        for r in results:
            print r
```

用下面的命令运行代码：

```
$ python -m cProfile -o search.prof search.py John
```

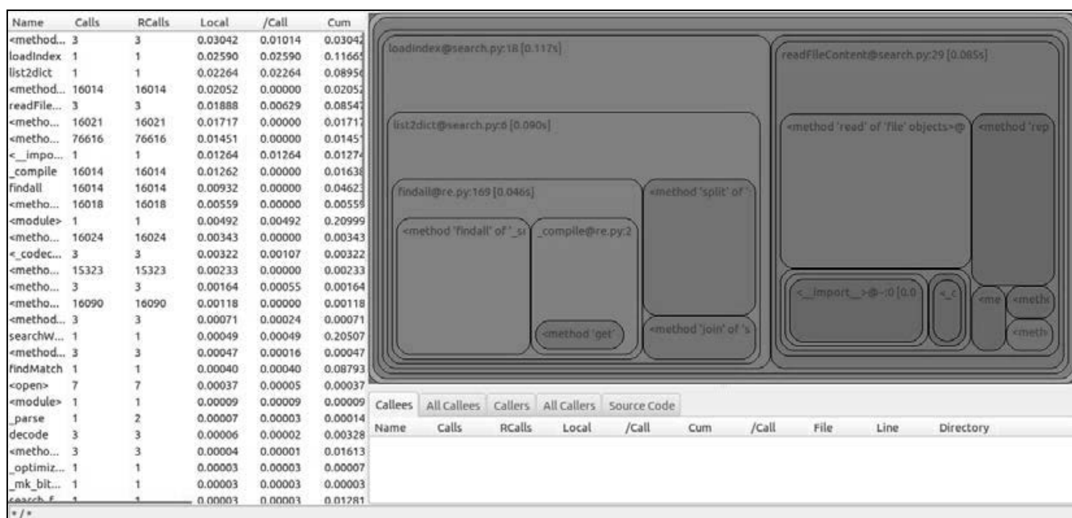
我们将获得的输出结果如下面的截图所示（假设我们在files文件夹里已经放了一些书）。

```
fernandodoglio@glb-l0841:~/workspace/writing/python$ python -m cProfile -o search.prof search.py John
John Burroughs her deepest debt is due To this clear-visioned prophet who has opened the blind eyes
John Burroughs  All the familiar woodpeckers have two characteristics most prominently exemplified i
John Burroughs who calls this bird "the wild Irishman of the flycatchers"   OLIVE-SIDED FLYCATCHER (
John Burroughs calls it "It is not a proud gorgeous strain like the tanager's or the grosbeak's" he
John Burroughs has called the bird the ''bush sparrow"   FOX SPARROW (Passerella ilica) Finch family
John Burroughs "It begins with the words fe-u fe-u fe-u and runs off into trills and quavers like th
John Burroughs in ever-delightful "Wake Robin"; "but no he is doomed to wear the name of some discov
John Burroughs calls him of all our birds "the most native and democratic"  How the robin dominates
John Burroughs is like scarlet "strong intense emphatic" but it is sweet and is more rapidly uttered
John Friese for making the drawings; and to the following for the use of the originals of the illust
John Burroughs  This eBook is for the use of anyone anywhere at no cost and with almost no restricti
John Burroughs  Commentator: Mary E Burt  Posting Date: January 17 2009 [EBook #3163] Release Date:
John Burroughs   With An Introduction  By Mary E Burt   And A Biographical Sketch   CONTENTS   Biogr
John Burroughs's birth A little before the day when the wake-robin shows itself that the observer mi
John Burroughs His books are redolent of the soil and have such "freshness and primal sweetness" tha
John Burroughs's essays I at once foresaw many a ramble with my pupils through the enchanted country
John Burroughs is to live in the woods and fields and to associate intimately with all their little
John Burroughs's essays is much healthier than the over-wrought dramatic action which sets all the n
John Burroughs more than almost any other writer of the time has a prevailing taste for simple words
John Burroughs   MARY E BURT  JONES SCHOOL CHICAGO Sept 1 1887     BIRDS     BIRD ENEMIES   How sure
John the Baptist during his sojourn in the wilderness his divinity school-days in the mountains and
John Burroughs  *** END OF THIS PROJECT GUTENBERG EBOOK BIRDS AND BEES ***  ***** This file should b
```

输出结果可以通过高亮关键词，或者显示更多的上下文来改进。不过我们把重点放到运行时

间的分析上。

下面我们看看在RunSnakeRun打开search.prof文件时显示的效果：



和前面的最小公倍数的示例相比，图中多了很多方块。然而，我们可以看看一眼能捕获哪些信息。

两个最耗时的函数是`loadIndex`和`list2dict`，紧随其后的是`readFileContent`函数。我们可以在左边列表中看到这些。

- 这些函数的大部分时间都消耗在调用的其他函数上了。因此，它们的总消耗时间很高，但是内部运行时间较低。
- 如果按照内部运行时间对列表排序，我们会看到排在最前面的5个函数是：
  - 读取文件对象的`read`方法
  - `loadIndex`函数
  - `list2dict`函数
  - 正则表达式对象的`findAll`方法
  - `readFileContent`函数

让我们先看看`loadIndex`函数。虽然大部分时间都花在了`list2dict`函数上，但我们仍然有一个小优化可以做，即可以简化代码以明显地降低代码的内部运行时间：

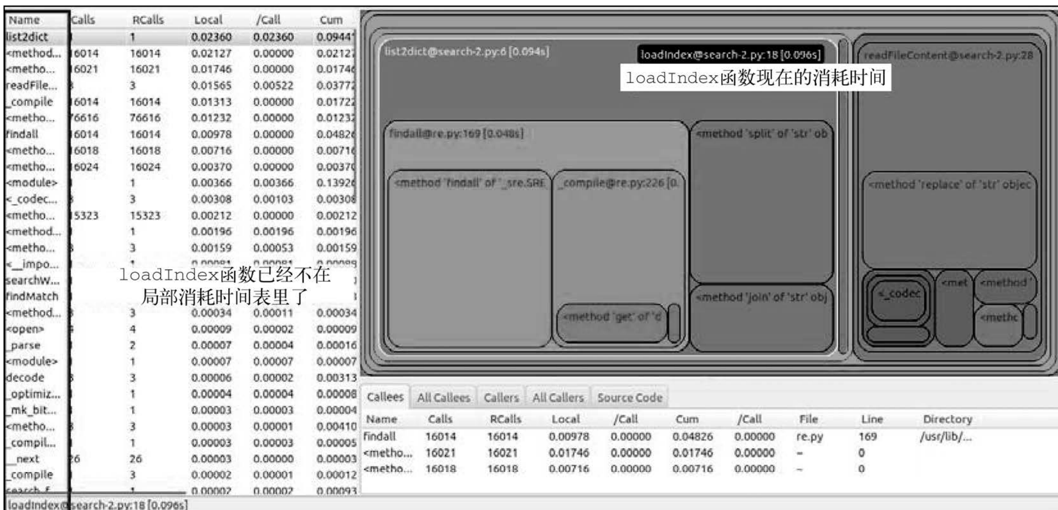
```
def loadIndex():
    indexFilename = "./index-file.txt"
    with open(indexFilename, 'r') as fh:
        # 我们不用循环遍历每一行加入数组，
```

```
        # 而是通过readlines按行读取文件内容
        indexLines = fh.readlines()
        index = list2dict(indexLines)
        return index
```

一点简单的改变就可以把内部运行时间从0.03秒降到了0.00002秒。虽然它不是一个大瓶颈，但是我们在优化性能的同时也改善了代码的可读性，可谓一举两得。

经过前面的分析我们知道函数消耗的时间大都花在了被调用的其他函数上了。所以函数的内部执行时间我们已经优化得差不多了，下面把精力集中到下一个函数上：`list2dict`。

但是在此之前，让我们先看看经过前面的小改进之后，性能的变化情况：



现在，让我们转移到`list2dict`函数上。这个函数的主要作用是把索引文件的每一行解析成可以使用的形式。更具体地说，就是它会把索引文件中的每一行解析成一个以单词为键的哈希表（或Python字典），使得我们搜索单词时，查询时间复杂度平均可以达到$O(1)$（如果不记得这些了，请查阅第1章）。字典的值是实际文件的路径，以及文件中单词所在位置的坐标。

经过分析，我们发现函数内部消耗的时间绝大多数都消耗在正则表达式上面。正则表达式是非常给力的工具，但是有时候我们可以用`split`和`replace`方法替换。所以，让我们看看在不使用正则表达式的情况下，如何解析数据并获取同样的结果，而且消耗的时间更少：

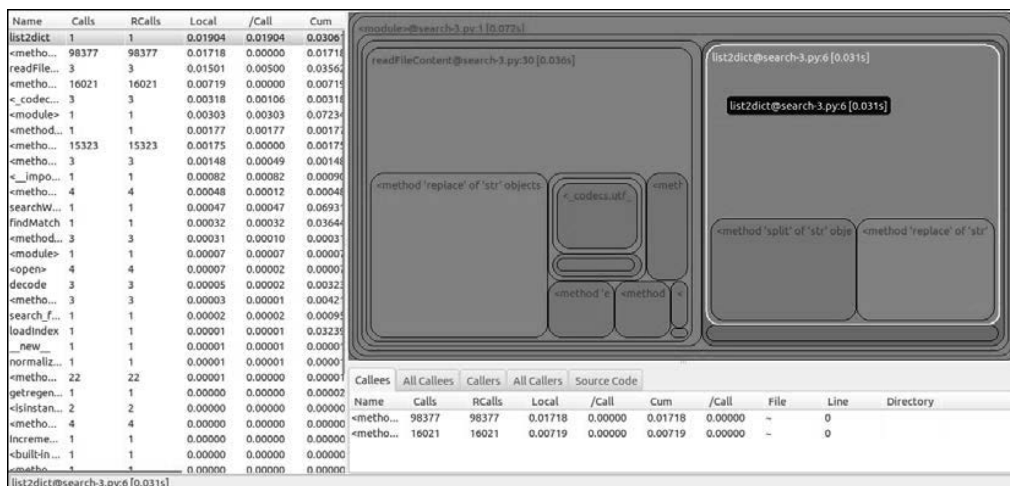
```
def list2dict(l):
    retDict = {}
    for item in l:
        lineParts = item.split(',(')
        word = lineParts[0]
        indexDataParts = [x.replace(")","") for x in lineParts[1:]]
        retDict[word] = indexDataParts
```

```
        return retDict
```

代码看起来更简洁了。代码里没有正则表达式（有时候这么做可以让代码更易读，因为并非每个人都能看懂正则表达式）。代码的行数更少了。我们去掉了join行，也去掉了丑陋累赘的del行代码。

但是，我们增加了一行列表综合代码，这行代码是对列表中的每个元素都应用replace方法。

现在让我们再看看下面的性能分析图：



这次变化很明显。如果你对比之前的两个截图，会发现list2dict函数从左边转移到了右边，这说明list2dict函数的运行时间比readFileContent函数要少。函数的内部结构也变得更简单，现在函数里只有split和replace函数。最后，让我们看看时间数据。

- 局部运行时间从0.024秒降到了0.019秒。显然局部时间并没有降低很多，因为我们在代码内部做了很多事情。时间降低主要是由于去掉了del行和join行。
- 总运行时间的下降比较显著，从0.094秒下降到了0.031秒，这是不使用复杂函数（正则表达式）的结果。

我们把总运行时间降到了原来的1/3。因此，这个函数的优化效果很好，尤其是当索引字典很大时，节省的时间会更多。

> 最后一个假设并非总是正确的，具体取决于所用算法的类型。但是，在示例中，我们遍历了索引文件的每一行，所以可以放心地做出这样的假设。

让我们简单对比一下原始代码和最后一次优化后代码的性能，看看程序总运行时间的改善

情况：

Name	Calls	RCalls	Local	/Call	Cum
	0	2	0.00000	0.00000	0.20999
<module>	1	1	0.00492	0.00492	0.20999
searchWord	1	1	0.00049	0.00049	0.20507
loadIndex	原始代码的运行时间		0.02590	0.02590	0.11665

Name	Calls	RCalls	Local	/Call	Cum
	0	2	0.00000	0.00000	0.07234
<module>	1	1	0.00303	0.00303	0.07234
searchW...	1	1	0.00047	0.00047	0.06931
findMatch	1	优化后代码的运行时间	0.00032	0.00032	0.03644

最后，你会发现，程序的总运行时间从大约0.2秒降到了0.072秒。

下面是代码的最终版本，所有的优化都放在里面了：

```
import sys

# 把索引文件中的项目转变成词典，
# 以单词为索引
def list2dict(l):
    retDict = {}
    for item in l:
        lineParts = item.split(',(')
        word = lineParts[0]
        indexDataParts = [x.replace(")","") for x in lineParts[1:]]
        retDict[word] = indexDataParts
    return retDict

# 把索引内容加入内存并解析
def loadIndex():
    indexFilename = "./index-file.txt"
    with open(indexFilename, 'r') as fh:
        # 我们不用循环遍历每一行以加入数组，
        # 而是通过readlines按行读取文件内容
        indexLines = fh.readlines()
        index = list2dict(indexLines)
        return index

# 读取文件内容，单词使用UTF-8编码格式，
# 移除不需要的单词（不希望出现在索引中的单词）
def readFileContent(filepath):
    with open(filepath, 'r') as f:
        return [x.replace(",", "").replace(".","").
replace("\t","").replace("\r","").replace("|","").strip(" ") for x in
f.read().decode("utf-8-sig").encode("utf-8").split( '\n' )]

def findMatch(results):
    matches = []
    for r in results:
        parts = r.split(',')
```

```
            filepath = parts[0]
            del parts[0]
            fileContent = ' '.join(readFileContent(filepath))
            for offset in parts:
                ioffset = int(offset)
                if ioffset > 0:
                    ioffset -= 1
                matchLine = fileContent[ioffset:(ioffset + 100)]
                matches.append(matchLine)
        return matches

# 在索引文件中搜索单词
def searchWord(w):
    index = None
    index = loadIndex()
    result = index.get(w)
    if result:
        return findMatch(result)
    else:
        return []

# 让用户自定义单词……
searchKey = sys.argv[1] if len(sys.argv) > 1 else None

if searchKey is None:  # 如果没有搜到，就打印一条信息
    print "Usage: python search.py <search word>"
else:  # 如果单词存在，则进行搜索
    results = searchWord(searchKey)
    if not results:
        print "No results found for '%s'" % (searchKey)
    else:
        for r in results:
            print r
```

3.3 小结

综上所述，我们在这一章介绍了两个最流行也是Python开发者最常用的可视化工具，将`cProfile`之类的性能分析器生成的结果变得直观易懂。我们用新的工具分析了旧代码，也分析了一些新代码。

从下一章开始，我们将介绍更具体的性能优化手段。我们将介绍一些在分析和优化代码性能时总结的实践经验，以及值得推荐给读者的好习惯。

第4章 优化每一个细节

Python的性能分析之路已经扬帆起航。运行性能分析程序只能发现问题，不能解决问题。在前几章学习性能分析器时，我们看过了一些示例，也做过了一些优化，但是都没有进行详细的解释。

这一章将详细介绍优化的过程。为此，我们得从基础知识开始讲起。我们将结合Python语言自身的特点去分析：这一章没有辅助工具，只有Python语言和正确使用Python的方法。

本章主题如下：

- 函数返回值缓存/函数查询表
- 默认参数的用法
- 列表综合表达式
- 生成器
- ctypes
- 字符串连接
- 其他Python优化技巧

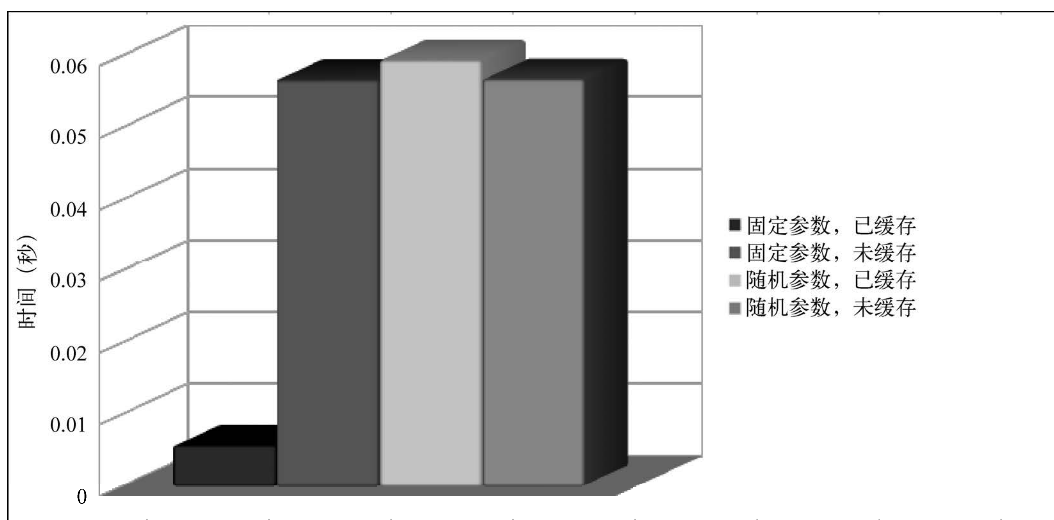
4.1 函数返回值缓存和函数查询表

函数返回值缓存（memoization）/函数查询表（lookup table）是优化一段代码（一个函数）最常用的手段。我们可以先把函数、输入参数和返回值全部都保存起来，在函数下次被调用时直接使用存储的结果（不需要重新计算所有数据）。因为这是一种函数返回值缓存技术，所以可能容易与缓存（caching）混淆，尽管这个术语也用于其他优化技术（比如HTTP caching、buffering等）。

其实这个方法非常强大，因为如果能正确地实现缓存，就可以把一个非常耗时的函数调用变成$O(1)$时间复杂度（关于时间复杂度的更多信息，请参考第1章）。把输入参数放入字典，作为不重复的键，可以在字典中保存函数结果，也可以在函数结果已经被保存之后通过字典的键查询直接获取。

当然，这种技术也是有代价的。如果我们要记住被缓存函数的所有信息，就需要拿内存空间换运行时间。如果函数需要的存储空间不超过系统内存容量，那么这个代价还是非常值得的。

这种优化方法的典型使用情形是在处理固定参数的函数被重复调用时。这样做可以确保每次函数被调用时，直接返回缓存结果。如果函数被调用很多次，但是参数是随机变化的，那么我们存储函数就没什么效果了。对比效果如下图所示。



你会发现最左侧的条形（固定参数，已缓存）显然是运行速度最快的，而其他三个条形的运行速度差不多。

产生上图结果的代码如下所示。为了获得函数的消耗时间，代码会在不同条件下调用twoParams和twoParamsMemoized函数几百次，然后把消耗时间写到日志里：

```
import math
import time
import random

class Memoized:
  def __init__(self, fn):
    self.fn = fn
    self.results = {}


  def __call__(self, *args):
    key = ''.join(map(str, args[0]))
    try:
      return self.results[key]
    except KeyError:
      self.results[key] = self.fn(*args)
```

```
    return self.results[key]


@Memoized
def twoParamsMemoized(values, period):
  totalSum = 0
  for x in range(0, 100):
    for v in values:
      totalSum = math.pow((math.sqrt(v) * period), 4) + totalSum
  return totalSum



def twoParams(values, period):
  totalSum = 0
  for x in range(0, 100):
    for v in values:
      totalSum = math.pow((math.sqrt(v) * period), 4) + totalSum
  return totalSum



def performTest():
    valuesList = []
    for i in range(0, 10):
        valuesList.append(random.sample(xrange(1, 101), 10))

    start_time = time.clock()
    for x in range(0, 10):
      for values in valuesList:
          twoParamsMemoized(values, random.random())
    end_time = time.clock() - start_time
    print "Fixed params, memoized: %s" % (end_time)

    start_time = time.clock()
    for x in range(0, 10):
      for values in valuesList:
          twoParams(values, random.random())
    end_time = time.clock() - start_time
    print "Fixed params, without memoizing: %s" % (end_time)


    start_time = time.clock()
    for x in range(0, 10):
      for values in valuesList:
          twoParamsMemoized(random.sample(xrange(1,2000), 10), random.random())
    end_time = time.clock() - start_time
    print "Random params, memoized: %s" % (end_time)

    start_time = time.clock()
    for x in range(0, 10):
      for values in valuesList:
          twoParams(random.sample(xrange(1,2000), 10), random.random())
    end_time = time.clock() - start_time
```

```
    print "Random params, without memoizing: %s" % (end_time)

performTest()
```



> 从前面的图形中可以获得的主要结论是，和编程技术不同的侧面一样，没有一种算法是解决所有问题的银弹[①]。虽然函数返回值缓存方法是优化代码的基础手段，但是显然它并不能优化所有条件下的代码。

对于代码本身，其实没什么内容。这是一个非常简单也不太实际的代码示例。performTest函数会把每个测试都运行10次，然后记录每次运行的时间。你会发现我们在这里没用性能分析器，仅仅是用简单且临时性的做法记录了函数运行时间，这就可以满足我们的需求。

为了实现缓存的目的，两个函数在运行数学函数时简单地用一组数字作为输入参数。

关于输入参数的另一个有趣的事情是，由于函数的第一个参数是一个数字列表，所以我们不能把它作为Memoized类results字典的args键[②]。于是我们用了下面这行代码处理输入：

```
key = ''.join(map(str, args[0]))
```

这行代码会把输入的列表连成一个字符串，作为字典的键。这里没有使用第二个参数，因为它是随机产生的，也就是说键永远不会重复。

上面这种函数返回值缓存方法的另一种版本是，在函数初始化阶段预先计算所有的值（当然，假设我们的输入是有限的），然后在执行时调用查询表。可以这么做的前提条件是：

- 输入值必须是有限的，否则无法预先算出所有值；
- 查询表带了所有值，必须满足内存限制；
- 和前面提到的一样，输入值至少要使用一次，这样优化才有意义，也值得我们多付出一些努力。

构建查询表的方法有好几种，针对不同的类型进行优化。具体选择哪种方法，是根据需要进行优化的问题和目标的类型决定的。下面介绍一些例子。

4.1.1 用列表或链表做查询表

这种方法是迭代无序列表，里面的每个元素对应字典的键，与要查找的函数结果对应。这显然是一种效率低下的方法，因为根据大O标记法，迭代列表的算法的平均与最坏情况下的时间复

① 参考《人月神话》。——译者注

② Python中的列表是可变类型，不能作为字典的键。——译者注

杂度都是$O(n)$。如果条件合适，这么做可以比每次调用函数运行结果更快一些。



在这种情况下，用链表可以实现比普通列表更高的性能。但是，链表的类型（双向链表，允许直接链接首尾元素的单向链表）也会对性能产生显著的影响。

4.1.2　用字典做查询表

这个方法是用一维字典进行查询，被索引的每个键都是输入条件的组合（足以组成一个唯一的键）。在某些情况下（像上面介绍的例子那样），这种查询方式可能是最快的，甚至比二分查找都要快（按照大O标记法，这种数据结构查找算法的时间复杂度是$O(1)$）。



需要注意的是，只要每次都能够生成不重复的键，这个算法就是有效的。但是，随着字典规模增大（哈希），碰撞频繁，性能会下降。

4.1.3　二分查找

这种方法只在列表是有序的前提下才可以使用。在值已经被排序的情况下，这种方法可以作为一种选择。但是，排序是需要消耗时间的，因此会影响整体性能。然而，二分查找即使是处理很长的列表，效率也很高（按照大O标记法，最差情况下的时间复杂度也是$O(\log n)$）。算法通过重复地判断被查询的值在列表中的哪一半，查找目标值或者确定值不在列表中。

综合上面介绍的内容，看看前面介绍过的`Memoized`函数，会发现我们是用字典实现查询表。但是，在别的案例中，可能需要用其他算法实现。

4.1.4　查询表使用案例

查询表优化的基本示例很多，但是最常见的可能就是优化三角函数。按照计算时间来看，这类函数运行得非常慢。在重复使用时，这些函数会对程序性能造成沉重负担。

这就是通常建议预先计算这类函数值的原因。对于输入参数有无穷可能的函数，这么做是可行的。因此，开发者不得不为了性能而牺牲计算精度，预先计算离散的整数值输入（就是从浮点数到整数）。

这种方法在某些既要求高性能又需要精度的环境中可能不太适用。因此，折中的做法是，在预先计算的结果之间进行插值计算。实践表明这种做法精度更高。即使这样做时的性能无法同直接用查询表时相比，但是比每次直接调用三角函数计算结果的性能要好很多。

让我们来看一些例子，例如下面的三角函数：

```
def complexTrigFunction(x):
    return math.sin(x) * math.cos(x)**2
```

我们来看看如何简单地实现无需绝对精确的预先计算方法，再通过插值计算获得更高的精度。

下面的代码将计算-1000到1000范围内（只有整数）的函数值，然后再计算一些浮点数（只在更小的范围进行）的函数值：

```
import math
import time
from collections import defaultdict
import itertools

trig_lookup_table = defaultdict(lambda: 0)


def drange(start, stop, step):
    assert(step != 0)
    sample_count = math.fabs((stop - start) / step)
    return itertools.islice(itertools.count(start, step), sample_count)


def complexTrigFunction(x):
    return math.sin(x) * math.cos(x)**2


def lookUpTrig(x):
    return trig_lookup_table[int(x)]


for x in range(-1000, 1000):
    trig_lookup_table[x] = complexTrigFunction(x)

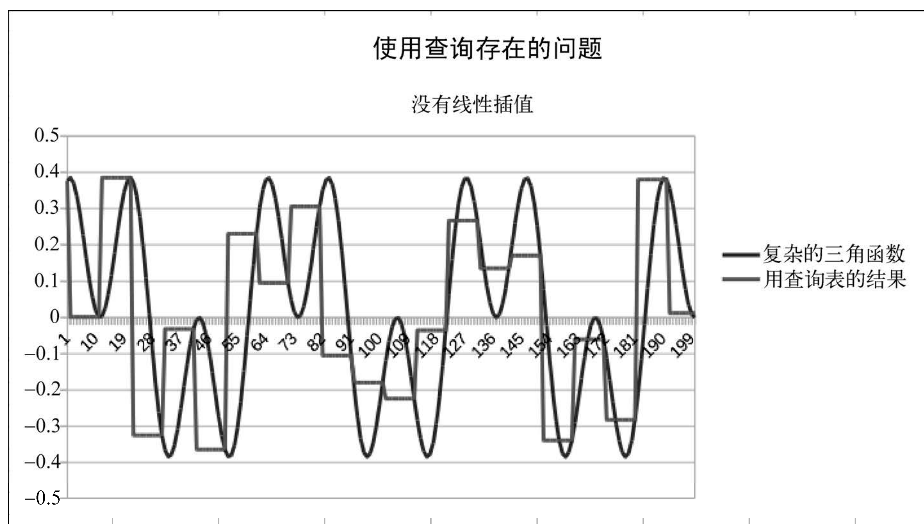
trig_results = []
lookup_results = []

init_time = time.clock()
for x in drange(-100, 100, 0.1):
    trig_results.append(complexTrigFunction(x))
print "Trig results: %s" % (time.clock() - init_time)

init_time = time.clock()
for x in drange(-100, 100, 0.1):
    lookup_results.append(lookUpTrig(x))
print "Lookup results: %s" % (time.clock() - init_time)

for idx in range(0, 200):
    print "%s\t%s" % (trig_results[idx], lookup_results[idx])
```

上面代码的结果将演示简单查询表是多么不精确（如下图所示）。但是它在速度上获得了弥补，原始函数的运行时间需要0.001526秒，而使用这个简单查询表只需要0.000717秒。



从上图可以看出，没有线性插值会导致精度很低。你会发现，即使两个图形看起来很相似，但是查询表和直接运行`trig`函数获得的精度是不同的。现在，让我们再看看这个问题。不过这一次我们增加了一个插值计算（我们将把范围限制在`-PI`到`PI`之间）。

```
import math
import time
from collections import defaultdict
import itertools

trig_lookup_table = defaultdict(lambda: 0)


def drange(start, stop, step):
    assert(step != 0)
    sample_count = math.fabs((stop - start) / step)
    return itertools.islice(itertools.count(start, step), sample_count)


def complexTrigFunction(x):
    return math.sin(x) * math.cos(x)**2


reverse_indexes = {}
for x in range(-1000, 1000):
    trig_lookup_table[x] = complexTrigFunction(math.pi * x / 1000)


complex_results = []
lookup_results = []

init_time = time.clock()
for x in drange(-10, 10, 0.1):
```

```
    complex_results.append(complexTrigFunction(x))
print "Complex trig function: %s" % (time.clock() - init_time)

init_time = time.clock()
factor = 1000 / math.pi
for x in drange(-10 * factor, 10 * factor, 0.1 * factor):
    lookup_results.append(trig_lookup_table[int(x)])
print "Lookup results: %s" % (time.clock() - init_time)

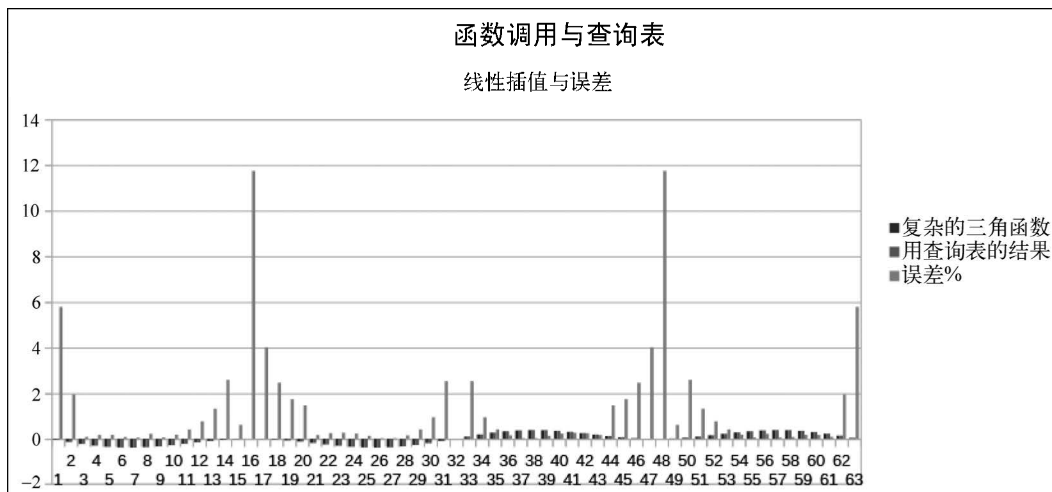
for idx in range(0, len(lookup_results)):
    print "%s\t%s" % (complex_results[idx], lookup_results[idx])
```

从上图中你应该已经注意到曲线是周期性的（这是因为我们把范围限制在-PI到PI之间）。因此我们将重点关注图形中的一部分数值。

上面脚本运行的结果也说明插值计算的方法比直接用三角函数计算结果更快，虽然比最初的简单查询表要慢一点。

插值函数	原始函数
0.000118秒	0.000343秒

下图和上一张图有些不同，尤其是因为图中用条形图显示了插值计算与真实值的相对误差。



最大的误差高达约12%（图中顶点所示）。但是，这发生在很小的数值上，比如–0.000852248551417与–0.000798905501416。因此这里的误差需要根据实际数值判断计算得是否合理。在我们的例子中，由于数值本身非常小，所以误差其实可以忽略不计了。



查询表还有其他应用场景，比如图像处理。但是，结合本书的主题，上面的例子应该已经足以体现查询表优化的优缺点了。

4.2　使用默认参数

还有一种优化技术与函数缓存技术不同，使用得并不十分普遍。这种方法是直接优化Python解释器的工作方式。

默认参数（default argument）可以在函数创建时就确定输入值，而不用在运行阶段才确定输入。



[这种方法只能用于在运行过程中参数不发生变化的函数和对象。]

下面用一个例子来演示这种优化手段如何使用。下面的代码里包括一个函数的两种版本，它们都随机计算三角函数：

```
import math

# 原始函数
def degree_sin(deg):
    return math.sin(deg * math.pi / 180.0)

# 优化的函数，因子变量是在函数创建时建立的，
# 所以直接用math.sin方法的查询值即可
def degree_sin(deg, factor=math.pi/180.0, sin=math.sin):
    return sin(deg * factor)
```



[如果文档没写清楚，这种优化方法是有问题的。因为在运行过程中，函数预先计算的项目是不能改变的，所以函数的接口容易造成混乱。]

通过一个简单快速的测试，我们就可以复核这种优化方法对函数的性能改善：

```
import time
import math


def degree_sin(deg):
    return math.sin(deg * math.pi / 180.0) * math.cos(deg * math.pi / 180.0)


def degree_sin_opt(deg, factor=math.pi/180.0, sin=math.sin, cos=math.cos):
    return sin(deg * factor) * cos(deg * factor)


normal_times = []
optimized_times = []

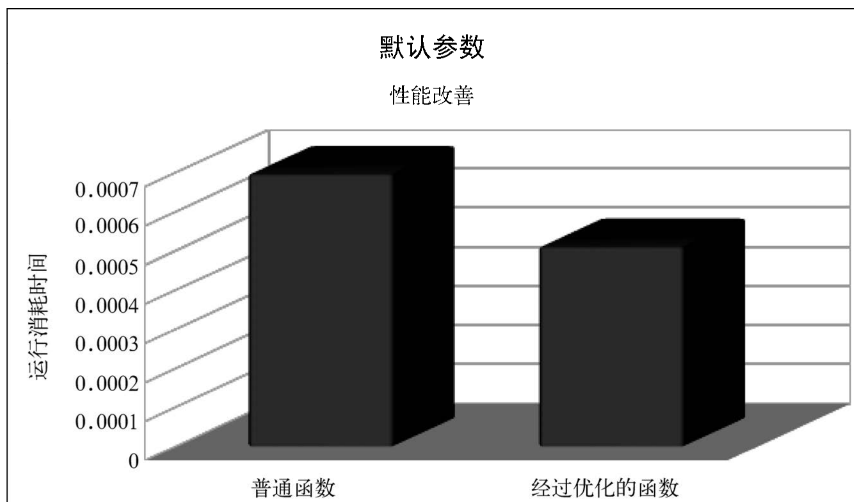

for y in range(100):
```

```
    init = time.clock()
    for x in range(1000):
        degree_sin(x)
    normal_times.append(time.clock() - init)

    init = time.clock()
    for x in range(1000):
        degree_sin_opt(x)
    optimized_times.append(time.clock() - init)


print "Normal function: %s" % (reduce(lambda x, y: x + y, normal_times, 0) / 100)
print "Optimized function: %s" % (reduce(lambda x, y: x + y, optimized_times, 0) / 100)
```

上面的代码统计了在0到1000范围内两个版本函数运行的时间。代码保存了测量结果，并最终得到了每个函数的平均运行时间。结果如下图所示。



显然这不是一个十分出色的优化手段。但是，它为我们节省了几毫秒的运行时间，因此值得我们关注。还要牢记的是，如果你在一个开发操作系统的团队中工作，使用这种方法会出问题。

4.3 列表综合表达式与生成器

列表综合表达式是Python提供的特殊范式，以数学方式表达要产生的列表，重点是描述计算什么，而不是描述计算的过程（经典的for循环语句）。

下面通过一个例子来更好地理解列表综合的运行方式：

```
# 用列表综合表达式产生0-100内的50个偶数
multiples_of_two = [x for x in range(100) if x % 2 == 0]
```

```
# 通过for循环产生同样的结果
multiples_of_two = []
for x in range(100):
    if x % 2 == 0:
        multiples_of_two.append(x)
```

列表综合并不是要完全取代for循环。在像上面的创建列表的例子中用for循环并没有问题。但是，因为for循环这种方式有副作用，所以不太推荐使用。用for循环可能得不到你需要的列表。你很可能在循环体中放了一些函数，并运行一些计算，但是最终并没有保存到列表中。这时，使用列表综合表达式可能会影响代码的可读性。

为了搞明白为什么列表综合表达式比for循环的性能更好，我们需要做一些代码分解工作，还得看一点儿字节码。代码之所以可以分解，是因为Python虽然是一个解释型语言，但是代码最终还是会编译成字节码。字节码需要经过处理才能被理解。因此，我们用dis模块把字节码转换成人能读懂的形式，然后分析它们执行的细节。

让我们看看下面的代码：

```
import dis
import inspect
import timeit

programs = dict(
    loop="""
multiples_of_two = []
for x in range(100):
    if x % 2 == 0:
        multiples_of_two.append(x)
""",
    comprehension='multiples_of_two = [x for x in range(100) if x % 2 == 0]',
)

for name, text in programs.iteritems():
    print name, timeit.Timer(stmt=text).timeit()
    code = compile(text, '<string>', 'exec')
    dis.disassemble(code)
```

代码输出结果包括两部分：

- 每段代码的运行时间
- 通过dis模块分解出的解释器指令集

输出结果如下面的截图所示（在你的运行结果中，时间可能会不同，但是其他内容应该是一样的）。

```
comprehension 7.48636889458
  1           0 BUILD_LIST               0
              3 LOAD_NAME                0 (range)
              6 LOAD_CONST               0 (100)
              9 CALL_FUNCTION            1
             12 GET_ITER
        >>   13 FOR_ITER                28 (to 44)
             16 STORE_NAME               1 (x)
             19 LOAD_NAME                1 (x)
             22 LOAD_CONST               1 (2)
             25 BINARY_MODULO
             26 LOAD_CONST               2 (0)
             29 COMPARE_OP               2 (==)
             32 POP_JUMP_IF_FALSE       13
             35 LOAD_NAME                1 (x)
             38 LIST_APPEND              2
             41 JUMP_ABSOLUTE           13
        >>   44 STORE_NAME               2 (multiples_of_two)
             47 LOAD_CONST               3 (None)
             50 RETURN_VALUE
loop 9.42489385605
  2           0 BUILD_LIST               0
              3 STORE_NAME               0 (multiples_of_two)

  3           6 SETUP_LOOP              52 (to 61)
              9 LOAD_NAME                1 (range)
             12 LOAD_CONST               0 (100)
             15 CALL_FUNCTION            1
             18 GET_ITER
        >>   19 FOR_ITER                38 (to 60)
             22 STORE_NAME               2 (x)

  4          25 LOAD_NAME                2 (x)
             28 LOAD_CONST               1 (2)
             31 BINARY_MODULO
             32 LOAD_CONST               2 (0)
             35 COMPARE_OP               2 (==)
             38 POP_JUMP_IF_FALSE       19

  5          41 LOAD_NAME                0 (multiples_of_two)
             44 LOAD_ATTR                3 (append)
             47 LOAD_NAME                2 (x)
             50 CALL_FUNCTION            1
             53 POP_TOP
             54 JUMP_ABSOLUTE           19
             57 JUMP_ABSOLUTE           19
        >>   60 POP_BLOCK
        >>   61 LOAD_CONST               3 (None)
             64 RETURN_VALUE
```

首先，上图中的结果说明列表综合版的代码确实比for循环要快。现在，让我们仔细地逐条对比两种方式的指令集列表，以便更好地理解两者的差异。

for循环指令	注　释	列表综合指令	注　释
BUILD_LIST		BUILD_LIST	
STORE_NAME	列表multiples_of_two的定义		
SETUP_LOOP			
LOAD_NAME	range函数	LOAD_NAME	range函数
LOAD_CONST	100（range函数的属性值）	LOAD_CONST	100（range函数的属性值）
CALL_FUNCTION	调用range函数	CALL_FUNCTION	调用range函数
GET_ITER		GET_ITER	
FOR_ITER		FOR_ITER	
STORE_NAME	临时变量x	STORE_NAME	临时变量x
LOAD_NAME		LOAD_NAME	
LOAD_CONST	x % 2 == 0	LOAD_CONST	x % 2 == 0
BINARY_MODULO		BINARY_MODULO	
LOAD_CONST		LOAD_CONST	
COMPARE_OP		COMPARE_OP	
POP_JUMP_IF_FALSE		POP_JUMP_IF_FALSE	
LOAD_NAME		LOAD_NAME	
LOAD_ATTR	查询append方法	LIST_APPEND	把值追加到列表中
LOAD_NAME	加载变量x的值		
CALL_FUNCTION	把值追加到列表中		
POP_TOP			
JUMP_ABSOLUTE		JUMP_ABSOLUTE	
JUMP_ABSOLUTE		STORE_NAME	
POP_BLOCK		LOAD_CONST	
LOAD_CONST		RETURN_VALUE	
RETURN_VALUE			

从上表可以看出，for循环产生的指令集更长。列表综合产生的指令集就像是for循环指令集的真子集，主要的差异是数值被增加到列表中的方式不同。在for循环里，数值是一个一个增加的，用到三个指令（LOAD_ATTR、LOAD_NAME和CALL_FUNCTION）。但是，从图中可以看出，列表综合只用了一个简单且已经经过优化的指令（LIST_APPEND）。



这就是为什么在处理列表时，for循环不能作为主力使用。这是因为列表综合不仅更高效，有时候还可以实现更好的代码可读性。

但是，即使for循环需要做额外操作（会产生副作用），也切记不要肆意将所有的for循环都改成列表综合。这是因为有时候未经优化的列表综合，可能比for循环消耗的时间更长。

最后，还有一个相关的知识点值得关注：在处理大列表的时候，列表综合表达式可能就不好使了。这是因为列表综合需要直接产生每一个值。因此，如果你要处理一个包含10万元素的列表，

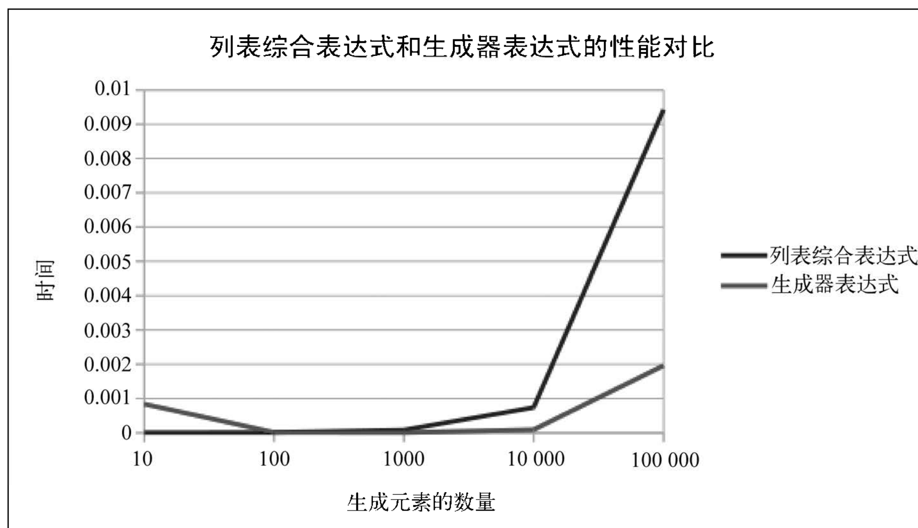
还有一种更好的办法。你可以用生成器表达式（generator expression），不需要直接返回列表，而是返回一个生成器对象，它的API与列表类似。但是，每当你请求列表元素时，生成器表达式就会为你动态地生成列表元素。

生成器对象和列表对象的主要差异是，生成器对象不支持随机接入（random access）。所以你不能再用方括号直接标记表达式。但是你可以通过`for`循环遍历生成器对象获得列表：

```
my_list = (x**2 for x in range(100))
# 你不能这么写
print my_list[1]
# 但是可以这么写
for number in my_list:
    print number
```

列表对象和生成器对象的另一个关键差异是，生成器对象只能遍历一次，而列表对象可以遍历任意次。这个差异非常重要，因为它直接影响你使用列表的效率。因此，在决定使用列表综合表达式还是生成器对象时，这一点不可忽略。

接入生成器的列表元素时可能会增加一点儿资源消耗，但是用它创建列表的速度更快。列表综合表达式和生成器表达式创建不同长度列表的时间如下图所示。



从上图可以看出，生成器表达式在创建规模较大的列表时更有优势，而创建规模较短的列表时，列表综合表达式更有优势。

4.4　ctypes

ctypes库可以让开发者直接进入Python的底层，借助C语言的力量进行开发。这个库只有官方版本解释器（CPython）里面才有，因为这个版本是C语言写的。其他版本，如PyPy和Jython，都不能接入这个库。

这个连接C语言的接口可以做很多事情，因为你可以直接加载预编译代码，并用C语言执行。也就是说，通过它你就可以接入Windows系统上的`kernel32.dll`和`msvcrt.dll`动态链接库，以及Linux系统上的`libc.so.6`库。

结合我们的性能优化目标，我们将介绍如何加载自定义C语言库，以及如何加载操作系统库来利用优化过的代码。关于这个库的具体用法，请参考官方文档：https://docs.python.org/2/library/ctypes.html。

4.4.1　加载自定义 ctypes

有时，无论我们在代码上用了多少优化方法，可能都没法儿满足我们对性能的要求。这时我们可以把关键代码写成C语言，编译成一个库，然后导入Python当作模块使用。

下面通过一个例子来演示如何实现这种优化，以及性能可以改善到何种程度。

需要解决的问题很简单，也是一个基本问题。我们的代码需要从100万个整数的列表中找出所有素数。

程序代码如下所示：

```
import math
import time


def check_prime(x):
    values = xrange(2, int(math.sqrt(x)))
    for i in values:
        if x % i == 0:
            return False

    return True


init = time.clock()
numbers_py = [x for x in xrange(1000000) if check_prime(x)]
print "%s" % (time.clock() - init)
```

上面的代码很简单。你也可以把列表综合表达式改成生成器进行改善。但是，为了演示C语言优化的效果，我们暂时不这样做。现在看看C代码，纯Python版的平均运行时间是4.5秒。

让我们写一个C语言版的check_prime函数，然后把它当作共享库（.so）导入Python代码中：

```
#include <stdio.h>
#include <math.h>

int check_prime(int a)
{
    int c;
    for ( c = 2 ; c <= sqrt(a) ; c++ ) {
        if ( a%c == 0 )
            return 0;
    }

    return 1;

}
```

用下面的命令产生库文件：

```
$gcc -shared -o check_primes.so -fPIC check_primes.c
```

然后我们在Python中写入两个版本的函数，比较运行时间，代码如下：

```
import time
import ctypes
import math

check_primes_types = ctypes.CDLL('./check_prime.so').check_prime

def check_prime(x):
    values = xrange(2, int(math.sqrt(x)))
    for i in values:
        if x % i == 0:
            return False

    return True


init = time.clock()
numbers_py = [x for x in xrange(1000000) if check_prime(x)]
print "Full python version: %s seconds" % (time.clock() - init)

init = time.clock()
numbers_c = [x for x in xrange(1000000) if check_primes_types(x)]
print "C version: %s seconds" % (time.clock() - init)
print len(numbers_py)
```

上面的代码输出结果如下：

纯Python版	C语言版
4.49秒	1.04秒

性能提升的效果非常好！运行时间从4.5秒降到了1秒。

4.4.2　加载一个系统库

有时，并不需要自己动手写C函数。系统的库文件可能已经为你准备好了。你需要做的就是导入库文件，调用函数。

让我们再看一个简单的示例。

下面这行代码生成100万个随机数，一共消耗了0.9秒：

```
randoms = [random.randrange(1, 100) for x in xrange(1000000)]
```

而下面这行代码，只需要0.3秒：

```
randoms = [(libc.rand() % 100) for x in xrange(1000000)]
```

运行两种方法并打印各自运行时间的完整代码如下：

```
import time
import random
from ctypes import cdll


libc = cdll.LoadLibrary('libc.so.6') # Linux系统
#libc = cdll.msvcrt # Windows系统

init = time.clock()
randoms = [random.randrange(1, 100) for x in xrange(1000000)]
print "Pure python: %s seconds" % (time.clock() - init)

init = time.clock()
randoms = [(libc.rand() % 100) for x in xrange(1000000)]
print "C version : %s seconds" % (time.clock() - init)
```

4.5　字符串连接

之所以把Python的字符串单独作为本章的一节内容，是因为Python的字符串和其他语言中的字符串不太一样。在Python里，字符串是不可变的（immutable），就是说一旦你创建了一个字符串，就不能再改变它的值。

这显然是一种让人无语的设计，因为我们经常要在字符串变量中进行连接（concatenation）或替换等操作。但是，让普通的Python开发者没有意识到的是，在这种设计（不可变的字符串）背后还有许多超乎想象的事情。

由于字符串是不可变的，每当我们要做任何改变字符串内容的操作时，其实都是创建了一个

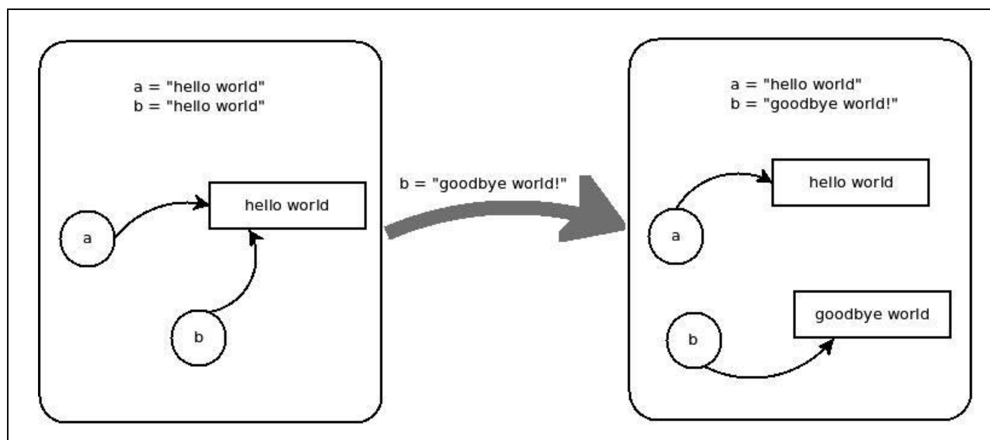
带有新内容的新字符串，我们的变量会指向新创建的字符串。因此，处理字符串时必须小心谨慎，三思而后行。

有一种非常简单的方法可以验证上面描述的场景。下面的代码创建了两个内容相同的字符串变量（我们定义每个变量时都写一次字符串）。然后，用id函数（在CPython里返回的是储存变量值的内存地址——指针）就可以比较两个变量。如果字符串是可变的，那么所有的对象都不一样，因此返回值也会不同。让我们看看代码：

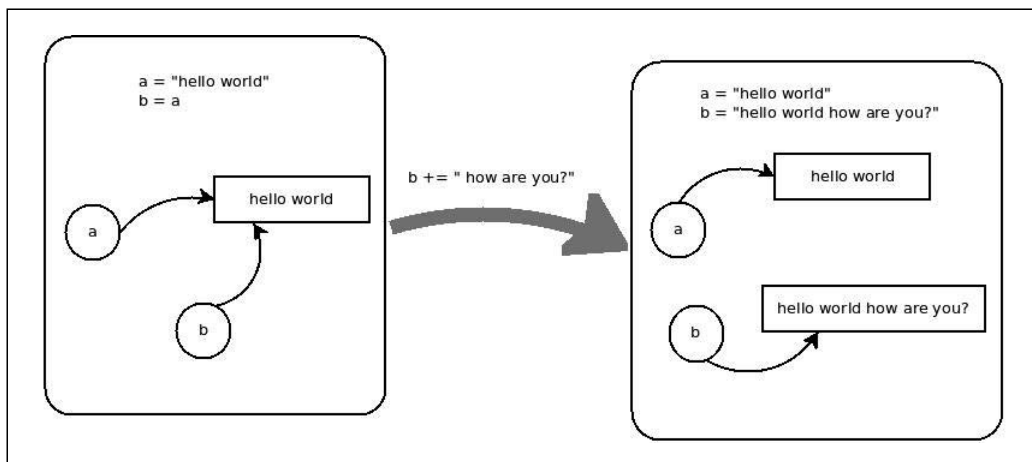
```
a = "This is a string"
b = "This is a string"
print id(a) == id(b) # 打印True
print id(a) == id("This is a string") # 打印True
print id(b) == id("This is another String") # 打印False
```

和代码中的注释一样，代码的输出结果是True、True和False，这其实显示了我们每次写This is a string字符串时，Python底层是如何重用字符串的。

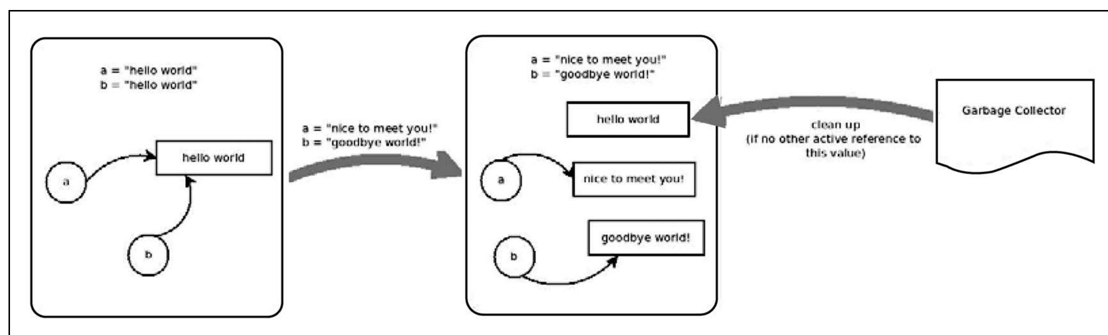
下图用一种图形化的方式表达了同样的含义。



虽然我们写了两次字符串，但是本质上两个变量都是指向同一块内存（里面包含实际的字符串）。如果我们把新的字符串赋值给其中一个变量，我们并不能改变字符串的内容，而是把变量指向了另一个内存地址。



之前的例子也发生了同样的事情，我们有一个变量`b`指向变量`a`。另外，如果我们想调整`b`，那么就要重新创建一个新的字符串。



最后，如果在上面的例子中，我们把两个变量的值都改变了，会发生什么事情呢？放在内存中的`hello world`字符串会怎么样呢？其实，如果这个字符串没有引用变量，那么垃圾回收器（Garbage Collector）就会过来回收它，并释放被占用的内存。

也就是说，不可变对象也不是一无是处。如果使用恰当，它们其实对性能有好处，例如，它们可以作为字典的键，甚至可以在不同的变量绑定（variable binding）之间进行共享（因为引用同一个字符串时其实都在用同一块内存）。也就是说，每当你使用字符串`hey there`时，其实都是完全一样的对象，无论它被赋值给了哪个变量（就像我们前面见到的那样）。

记住这些之后，想想一些常见情况下可能发生的事情，比如下面这个例子：

```
full_doc = ""
for word in word_list:
    full_doc += word
```

上面的代码用word_list列表中的每个元素串成了一个新字符串full_doc。这其实并非有效的内存使用方法，对吗？当我们想将不同片段重新组合成新字符串时，这其实是十分普遍的情景。有一种更加高效、更省内存的处理方式：

```
full_doc = "".join(world_list)
```

并且，这种方法更加容易阅读，书写更方便，内存与时间消耗都更少。下面的代码显示了每种做法消耗的时间。使用正确的命令，我们还可以看到for循环使用的内存要多一点儿：

```
import time
import sys

option = sys.argv[1]

words = [str(x) for x in xrange(1000000)]

if option == '1':
    full_doc = ""
    init = time.clock()
    for w in words:
        full_doc += w
    print "Time using for-loop: %s" % (time.clock() - init)

else:
    init = time.clock()
    full_doc = "".join(words)
    print "Time using join: %s" % (time.clock() - init)
```

通过下面使用Linux的time功能的命令行，我们可以执行脚本并测量内存使用量。

- for循环版本：

```
$ /usr/bin/time -f "Memory: %M bytes" python script.py 1
```

- join版本：

```
$ /usr/bin/time -f "Memory: %M bytes" python script.py 0
```

for循环版本的命令的输出结果如下：

```
Time using for-loop: 0.155635 seconds
Memory: 66212 bytes
```

join版本命令的输出结果如下：

```
Time using join: 0.015284 seconds
Memory: 66092 bytes
```

显然join版本消耗的时间更短，而且占用的内存（通过Linux系统的time命令）也更少。

在处理Python字符串时还会遇到的场景，就是连接不同类型的字符串；当你处理几个变量时

会遇到这样情况，就像下面这个例子：

```
document = title + introduction + main_piece + conclusion
```

每当你让系统实现新连接时，最终都要创建几个子字符串。因此更合理也更高效的做法，是用C语言字符串的变量内插法（variable interpolation）：

```
document = "%s%s%s%s" % (title, introduction, main_piece, conclusion)
```

另外，用`locals`函数创建子字符串的做法更好：

```
document = "%(title)s%(introduction)s%(main_piece)s%(conclusion)s" % locals()
```

4.6　其他优化技巧

前面介绍的都是程序优化过程中最常用的一些技术。有一些是专门面向Python的（比如字符串连接和ctypes），还有一些是通用的优化技术（比如函数返回值缓存和函数查询表）。

还有一些专门针对Python的小窍门，下面我们将逐一介绍。它们可能不会显著地提升性能，但是可以揭示Python语言的内部运作方式。

- **成员关系测试**：当我们想判断一个值是否在一个列表（list）中时（这里用的单词“list”具有一般性，并非特指Python的列表`list`），比如像“a in b”这样的操作，用集合（`set`）或字典（`dict`）（查找时间复杂度是$O(1)$）可以获得比列表（`list`）和元组（`tuple`）更好的性能。
- **不要重复发明轮子**：Python的标准库核心组件大都是用经过优化的C语言写成的。因此不需要你自建，而且你自建的很可能会更慢。像列表、元组、集合和字典这些数据类型，以及数组（`array`）、迭代工具（`itertools`）和队列（`collections.deque`）这些模块都推荐使用。使用内置函数，如`map(operator.add, list1, list2)`，也会比`map(lambda x, y: x+y, list1, list2)`更快。
- **不要忘记了队列**：当需要一个固定长度的数组或可变长度的栈（stack）时，列表非常适合。但是，在处理`pop(0)`或`insert(0, your_list)`操作时，可以试试`collections.deque`，因为它在列表的任何一端都可以快速地完成（$O(1)$）插入和弹出操作。
- **有时不定义函数更好**：调用函数会增加大量的资源消耗（我们之前已经看见过）。因此，有时候，尤其是在时间密集的循环体中内联函数代码，不调用外部函数，可以更加高效。这个窍门有一个很大的代价，因为它可能会损害代码的可读性和维护便利性。因此，仅当必须提升性能时才应这样做。下面的简单示例演示一个简单的查询操作如何增加了大量的运行时间：

```
import time
```

```
def fn(nmbr):
  return (nmbr ** nmbr) / (nmbr + 1)
nmbr = 0
init = time.clock()
for i in range(1000):
  fn(i)
print "Total time: %s" % (time.clock() - init)

init = time.clock()
nmbr = 0
for i in range(1000):
  nmbr = (nmbr ** nmbr) / (nmbr + 1)
print "Total time (inline): %s" % (time.clock() - init)
```

- **尽可能用key函数排序**：在对列表进行自定义规则的排序时，不要用比较函数排序，而是应该尽可能地用key函数排序。这是因为每个元素只需要调用一次key函数，而在运行过程中每一项都要调用比较函数好几次。让我们看看下面对比两种方法的例子：

```
import random
import time

# 创建两个随机数组
list1 = [[random.randrange(0, 100), chr(
  random.randrange(32, 122))] for x in range(100000)]
list2 = [[random.randrange(0, 100), chr(
  random.randrange(32, 122))] for x in range(100000)]

# 通过比较函数cmp()对两个数据排序
init = time.clock()
list1.sort(cmp=lambda a, b: cmp(a[1], b[1]))
print "Sort by comp: %s" % (time.clock() - init)  # 打印结果0.213434

# 把字符串元素作为词典的键，进行排序
init = time.clock()
list2.sort(key=lambda a: a[1])
print "Sort by key: %s" % (time.clock() - init)  # 打印结果0.047623
```

4

- **1比True好**：Python 2.3中的`while 1`得到了优化，跳转一次就能完成，而`while True`并没有，因此需要跳转好几次才能完成。因此用`while 1`比`while True`要更高效，虽然和内联函数一样，这么写也要付出很大代价。
- **多元赋值（multiple assignments）很慢但是……**：多元赋值（`a,b = "hellothere", 123`）通常比单独赋值要慢。但是，在进行变量交换时，它比普通方法要快（因为我们不需要使用临时变量和赋值过程）：

```
a = "hello world"
b = 123
# 这种做更快
a, b = b, a
# 相比这种方式
tmp = a
```

```
a = b
b = tmp
```

- **推荐使用链式比较**：在比较三个变量时，不要用$x<y$和$y<z$，可以用$x<y<z$。这样更容易阅读（更自然）且运行更快。
- **用命名元组（namedtuple）替换常规对象**：使用常规的类（`class`）方法创建存储数据的简单对象时，实例中会有一个字典存储属性。这个存储对属性少的对象来说是一种浪费。如果你需要创建大量的简单对象，会浪费大量内存。这种情况下，你可以用命名元组。这是一个新的`tuple`子类，可以轻松地构建并优化任务。关于命名元组的详细内容，请参考官方文档https://docs.python.org/2/library/collections.html#collections. namedtuple。下面的代码用常规类和命名元组分别创建了100万个对象，然后显示两种方式的消耗时间。

```
import time
import collections

class Obj(object):

  def __init__(self, i):
      self.i = i
      self.l = []

all = {}
init = time.clock()
for i in range(1000000):
  all[i] = Obj(i)
# 打印Regular对象：2.384832
print "Regular Objects: %s" % (time.clock() - init)

Obj = collections.namedtuple('Obj', 'i l')

all = {}
init = time.clock()
for i in range(1000000):
  all[i] = Obj(i, [])
# 打印NamedTuples对象：1.272023
print "NamedTuples Objects: %s" % (time.clock() - init)
```

4.7　小结

在这一章，我们介绍了一些优化技术。其中一些技术可以大幅提升程序的运行速度，或节省大量的内存，另外一些则只能略微提升程序的运行速度。这一章的大部分内容都是面向Python的技术，但是有一些技术也适用于其他编程语言。

下一章，我们将继续探索优化技术，重点介绍多线程和多进程技术，以及每种技术的应用场景。

第 5 章

多线程与多进程

在人们讨论代码优化时，并发（concurrency）和并行（parallelism）是两个不可能绕过的话题。但是，针对Python讨论这类话题时，通常都是在批评这门语言。批评者通常抱怨在Python中为并行和并发付出的努力与获得的真实效果（有些时候甚至没有效果）不成正比。

在这一章，我们将会看到批评者对Python的批评有时候是对的，有时候则并不正确。和大多数工具一样，这些技术都需要一些条件才能为开发者解决问题，而不是故意不让人使用。在介绍Python如何实现并行任务的过程中，真正值得介绍的两个主题如下。

- 多线程（multithreading）：这是实现真正的并行任务的经典做法。像C++和Java之类的语言也提供了这类特性。
- 多进程（multiprocessing）：虽然并非主流，而且有一些痛点需要解决，但是我们将会发现，它可以作为多进程的另一种版本。

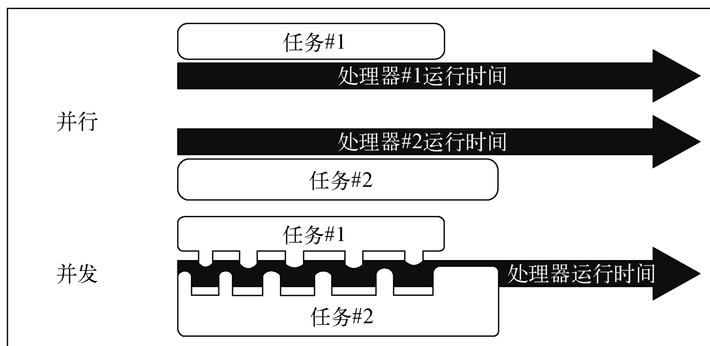
学读完本章后，你就会对多进程和多线程的差异有清晰的认识。另外，你还将明白什么是GIL（Global Interpreter Lock，全局解释锁），以及它如何影响你做出合理的并行选择。

5.1 并行与并发

这两个术语经常同时出现，并且被混用，但它们其实是完全不同的概念。并行是指两个或多个进行同时运行。这在多核心平台上可以实现，比如每个处理器上运行一个进程。

并发是指在同一个处理器上运行多个进程。在操作系统中，通常使用时隙（time slicing）技术来解决这类问题。但是，这种办法并非真正的并发，只是由于处理器切换任务的速度非常快，看起来像是并行。

并行与并发的差异如下图所示。



并发是现代操作系统常用的技术。因为这种技术与计算机处理器的数量无关，所以操作系统需要同时运行多个系统任务，而且还要满足用户的任何需求。为了解决这类问题，操作系统首先需要时刻关注处理器的任务调度时间表，记录每个任务需要的运行时间，然后在不同任务之间进行上下文切换，给每个任务一个时隙。

现在让我们回到本章一开始提到的问题：如何在Python程序中实现并行或并发呢？这正是多线程和多进程的作用所在。

5.2　多线程

多线程是程序在同样的上下文中同时运行多条线程的能力。这些线程共享一个进程的资源，可以在并发模式（单核处理器）或并行模式（多核处理器）下执行多个任务。

编写多线程的代码并不简单。但是，多线程可以提供一些非常显著的优点。

- **持续响应**：在单线程的程序中，执行一个长期运行的任务可能会导致程序冻结。多线程可以把这个长期运行的任务放在一个工作线程（worker thread）中，在程序并发地运行任务时可以持续响应客户需求。
- **更快的执行速度**：在多核处理器或多处理器的操作系统上，多线程可以通过真正的并行提高程序的运行速度。
- **较低的资源消耗**：利用线程模式，程序可以利用一个进程的资源响应多个请求。
- **更简单的状态共享与进程间通信机制**：由于线程都共享同一资源和内存空间，因此线程之间的通信比进程间通信简单。
- **并行化**：多核与多处理器系统可以实现多线程的每个线程独立运行。英伟达（Nvidia）的CUDA（Compute Unified Device Architecture，统一计算设备架构，http://www.nvidia.com/object/cuda_home_new.html），或科纳斯组织（Khronos Group）的OpenCL（https://www.khronos.org/opencl/），都是利用数十乃至数百个处理器进行并行计算的GPU运算环境。

多线程也有一些缺点。

- **线程同步**：由于多个线程是在同一块数据上运行的，所以需要引入一些机制预防竞态条件（race condition，会导致数据读取失败）。
- **问题线程导致集体崩溃**：虽然多个线程好像是独立运行的，但是一旦某个线程出现问题，就可能造成整个进程崩溃。
- **死锁**（deadlock）：这是线程操作的常见问题。通常，线程执行任务时会锁住正在使用的资源。当一个线程开始等待另一个线程释放资源，而另一个线程同时也要等待第一个线程释放资源时，就发生了死锁。

通常，多线程技术完全可以在多处理器系统上实现并行计算。但是，Python的官方版本（CPython）有一个GIL限制。GIL会阻止多个线程同时运行Python的字节码，这其实就不是真正的并行了。如果你的系统有4个处理器，多线程可以把CPU跑到400%，然而，你能看到的其实是100%甚至更慢点儿，这都是线程问题造成的。

> 需要注意的是，GIL并非只是Python（或CPython）的问题，其他编程语言也有，例如Ruby的官方版本Ruby MRI以及OCaml（https://ocaml.org/）。

CPython的GIL是有必要的，因为CPython的内存管理不是线程安全的。因此，为了让每个任务都按顺序进行，它需要确保运行过程中内存不被干扰。它可以更快地运行单线程程序，简化C语言扩展库的使用方法，因为它不需要考虑多线程问题。

但是，GIL是可以用一些办法绕过的。例如，由于GIL只阻止多个线程同时运行Python的字节码，所以你可以用C语言写程序，然后用Python封装。这样，在程序运行过程中GIL就不会干扰多线程并发了。

另一个GIL不影响性能的示例就是网络服务器了，服务器大部分时间都是在读数据包。这种情况下，增加线程可以读取更多的包，虽然这并不是真正的并行。这样做可以增加服务器的性能（每秒钟可以服务更多的客户），但是不会影响运行速度，因为每个线程都可以运行同样多的时间。

5.3 线程

现在，让我们介绍一些关于Python线程的知识，以便理解如何使用它们。它们由开始、执行序列和结论三部分构成。还有一个指令指针，用来跟踪正在执行的线程上下文。

指针可以在需要停止线程的时候置空或者中断。另外，它还可以临时性地保持不变。这时线程基本处于休眠状态。

为了在Python里使用线程，可以采用下面两种方法。

- **thread模块**：这个模块提供了有限的线程能力。它很容易使用，适合小项目，不过也会增加一点额外的资源消耗。
- **threading**：这是从Python 2.4引入的新模块，提供了一些更强大、高层次的线程支持。

5.3.1　用 thread 模块创建线程

虽然我们的重点是介绍threading模块，但是我们用一个例子快速地演示thread模块的用法是多么简单，它不需要写一大堆代码。

thread模块（https://docs.python.org/2/library/thread.html）提供了start_new_thread方法。我们可以向里面传入以下参数。

- 我们可以传入一个目标函数，里面包含我们要运行的代码。一旦函数返回值，线程就停止运行。
- 我们也可以传入一组（元组）参数，这组参数是目标函数的输入参数。
- 最后，我们还可以传入一个可选的命名参数词典。

下面来看一个例子：

```
#!/usr/bin/python
import thread
import time

# 打印时间5次，每次延迟“delay”秒
def print_time(threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s: %s" % (threadName, time.ctime(time.time()))

# 创建两个线程
try:
    thread.start_new_thread(print_time, ("Thread-1", 2, ))
    thread.start_new_thread(print_time, ("Thread-2", 4, ))
except:
    print "Error: unable to start thread"

# 我们需要让程序持续运行，否则线程就会消失
while 1:
    pass
```

上面代码打印的输出结果如下图所示。

```
Thread-1: Thu Mar 26 16:51:52 2015
Thread-1: Thu Mar 26 16:51:54 2015
Thread-2: Thu Mar 26 16:51:54 2015
Thread-1: Thu Mar 26 16:51:56 2015
Thread-2: Thu Mar 26 16:51:58 2015
Thread-1: Thu Mar 26 16:51:58 2015
Thread-1: Thu Mar 26 16:52:00 2015
Thread-2: Thu Mar 26 16:52:02 2015
Thread-2: Thu Mar 26 16:52:06 2015
Thread-2: Thu Mar 26 16:52:10 2015
```

上面的代码很简单，图中的运行结果清晰地展示两个线程是并发执行的。在`print_time`函数里面写了一个循环。如果连续运行代码两次，两次结果显示会间隔5秒钟。

但是，使用线程和不用线程其实没什么区别，我们其实是并发地运行两次循环。

这个模块还提供了一些容易使用的线程原生接口。示例如下：

`interrupt_main`

这个方法可以通过键盘向主线程发送中断异常。这就像是程序运行时用CTRL+C一样。如果没有收到中断异常，线程会发送下面的信号中断程序：

`exit`

这个方法会从后台退出程序。它的优点是中断线程时不会引起其他异常。让我们把之前的`print_time`改成下面的形式：

```
def print_time(threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s: %s" % (threadName, time.ctime(time.time()))
        if delay == 2 and count == 2:
            thread.exit()
```

运行结果如下所示：

```
Thread-1: Fri Mar 27 10:53:41 2015
Thread-2: Fri Mar 27 10:53:43 2015
Thread-1: Fri Mar 27 10:53:43 2015
Thread-2: Fri Mar 27 10:53:47 2015
Thread-2: Fri Mar 27 10:53:51 2015
Thread-2: Fri Mar 27 10:53:55 2015
Thread-2: Fri Mar 27 10:53:59 2015
```



当这两个语句执行完毕后，
该线程退出

`allocate_lock`方法可以为线程返回一个线程锁。这个锁可以帮助开发者保护重要代码在运行过程中不受竞态条件的干扰。

返回的线程锁对象有三个简单的方法。

- acquire：这个方法的主要作用是为当前的线程请求一把线程锁。它接受一个可选的整型参数。如果参数是0，那么进程锁一旦被请求则立即被获取，不需要任何等待。如果参数不是0，那么表示线程可以无条件地获取锁（如同不使用参数）。也就是说，如果线程需要等待获取锁，那么它可以等待。
- release：这个方法会释放线程锁，让下一个线程使用它。
- locked：如果线程锁被某个线程获取，就返回TRUE；否则返回FALSE。

下面用一个简单的示例演示加锁如何帮助多线程代码。这段代码用10个线程增加一个全局变量的值。因此，运行后全局变量的结果应该是10：

```
#!/usr/bin/python
import thread
import time

global_value = 0


def run(threadName):
    global global_value
    local_copy = global_value
    print "%s with value %s" % (threadName, local_copy)
    global_value = local_copy + 1


for i in range(10):
    thread.start_new_thread(run, ("Thread-" + str(i), ))

# 我们需要让程序持续运行，否则线程就会消失
while 1:
    pass
```

前面代码运行结果如下：

```
Thread-1 with value 0Thread-3 with value 0
 Thread-6 with value 0
Thread-8 with value 0

Thread-4 with value 0Thread-0 with value 1
Thread-5 with value 2

 Thread-2 with value 0
Thread-7 with value 0
Thread-9 with value 1
```

我们不仅没有正确地增加全局变量的值（只得到了2），而且打印字符串也有问题。可以看到，我们有两个字符串在一行，而它们原本应该是两行。两个字符串之所以打印在同一行，是因为两个线程同时执行打印操作。于是在同一时间，两个字符串出现在一行里。

同样的情况也发生在全局变量身上。当线程1、3、6、8、4、2和7读取全局变量的值以便加

1时，读到的值都是0（就是每个线程复制到local_value变量的值）。我们需要确保代码在复制数值、增加数值、打印数值的时候是被保护的（处于加锁的状态中），就是没有两个线程可以同时运行。要实现这个目标，我们将使用锁对象的两个方法：获取和释放锁。

示例代码如下：

```
#!/usr/bin/python
import thread
import time

global_value = 0


def run(threadName, lock):
    global global_value
    lock.acquire()
    local_copy = global_value
    print "%s with value %s" % (threadName, local_copy)
    global_value = local_copy + 1
    lock.release()


lock = thread.allocate_lock()

for i in range(10):
    thread.start_new_thread(run, ("Thread-" + str(i), lock))

# 我们需要让程序持续运行，否则线程就会消失
while 1:
    pass
```


现在，输出结果就合理了：

```
Thread-1 with value 0
Thread-6 with value 1
Thread-7 with value 2
Thread-3 with value 3
Thread-5 with value 4
Thread-2 with value 5
Thread-4 with value 6
Thread-0 with value 7
Thread-9 with value 8
Thread-8 with value 9
```

输出结果清晰，打印格式也正常，我们成功地实现了全局变量的递增。这两处改善都是通过线程锁机制解决的。当增加全局变量global_value的数值时，线程锁会阻止其他线程（没有获得锁的线程）执行这部分代码（把global_value变量的值读取到局部变量中，增加1）。因此，当线程锁是激活状态时，只有获取锁的线程能够运行这几行代码。当锁释放之后，线程队列中下一个线程才进行同样的操作。前一行代码会返回当前线程的标识：

```
get_ident
```

这是一个非0整数，没有其他含义，就是表示当前线程列表中的活动线程。这个整数会在一个线程结束或退出时被收回，因此在程序的生命周期中它不是唯一的。下面的代码在创建新线程时设置或返回线程栈的容量：

```
stack_size
```

这个参数是可选项（“这个”表示栈的容量）。这个容量可以是0，或者至少是32.768（32KB）。由操作系统决定，设置栈的容量还可能有其他限制。因此，在使用这个参数之前，要查看一下操作系统说明书。

> 虽然Python 3不在本书的讨论范围之内，但是thread模块在Python 3中已经改名为_thread。

5.3.2　用 threading 模块创建线程

这是目前Python中处理线程时普遍推荐的模块。这个模块提供了更完善、更高级的接口。不过它也增加了代码的复杂性，因为简单的_thread模块现在没有了。

这种情况符合Uncle Ben的名言：

> 越强大越复杂。（With great power comes great complexity.）

开个玩笑，其实threading模块是把线程的内容封装在一个类中，这样我们实例化这个类就可以使用线程。

我们可以创建一个这个模块提供的Thread类的子类（https://docs.python.org/2/library/thread.html）。另外。我们还可以在处理非常简单的问题时直接实例化这个类。让我们看看前面的示例如何转换成threading模块的形式：

```
#!/usr/bin/python
import threading
import time

global_value = 0

def run(threadName, lock):
    global global_value
    lock.acquire()
    local_copy = global_value
    print "%s with value %s" % (threadName, local_copy)
    global_value = local_copy + 1
    lock.release()
```

```
lock = threading.Lock()

for i in range(10):
    t = threading.Thread(target=run, args=("Thread-" + str(i), lock))
    t.start()
```

对于更复杂的情况，如果要更好地封装线程的行为，我们可能需要创建自己的线程类。

当使用子类方法写自己的线程类时，有一些事情是需要考虑的。

- ❑ 它们需要扩展threading.Thread类。
- ❑ 它们需要改写run方法，也可以使用__init__方法。
- ❑ 如果你改写构造器，需要在一开始调用父类的构造器（Thread.__init__）。
- ❑ 当线程的run方法停止或抛出未处理的异常时，线程将停止，因此要提前设计好方法。
- ❑ 可以用构造器方法的name参数命名你的线程。

即使你要重写run方法，里面包含了线程的主要逻辑，在线程的方法被调用时你也不能掌控它。不过你可以调用start方法，这个方法将创建一个新线程，然后在上下文中调用run方法。

下面让我们看一个简单的示例，它演示了线程处理中的一个常见问题。

```
import threading
import time


class MyThread(threading.Thread):

    def __init__(self, count):
        threading.Thread.__init__(self)
        self.total = count

    def run(self):

        for i in range(self.total):
            time.sleep(1)
            print "Thread: %s - %s" % (self.name, i)


t = MyThread(4)
t2 = MyThread(3)

t.start()
t2.start()

print "This program has finished"
```

代码输出结果如下所示：

```
This program has finished
Thread: Thread-2 - 0
 Thread: Thread-1 - 0
Thread: Thread-2 - 1
 Thread: Thread-1 - 1
Thread: Thread-2 - 2
 Thread: Thread-1 - 2
Thread: Thread-1 - 3
```

注意上图中高亮的部分，程序在其他内容出现之前先发送退出消息。在这里这并不是什么大问题。但是，在下面这种情况下就有问题了：

```
#....
f = open("output-file.txt", "w+")
t = MyThread(4, f)
t2 = MyThread(3, f)

t.start()
t2.start()
f.close() # 关闭文件处理器
print "This program has finished"
```



注意上面的代码会报错，因为在线程使用文件之前，会先关闭文件处理器。如果我们想避免这种情况，就需要使用join方法，这样就可以中断线程调用，直到目标线程运行结束为止。

在我们的示例中，如果从主线程中调用join方法，它可以保证在两个线程执行完成之前，程序不会运行主线程命令。我们需要确保在两个线程都启动之后再使用join方法。但是，我们可以按顺序依次结束它们：

```
#...
t.start()
t2.start()
# 两个线程同时运行，停止主线程
t.join()
t2.join()
f.close() # 两个线程都已经完成，关闭文件处理器
print "This program has finished"
```

这个方法还支持一个可选参数：时限（浮点数或None），以秒为单位。但是，join方法的返回值是None。因此，要检查操作是否已超时，需要在join方法返回之后检查线程是否还处于激活状态。如果线程是激活状态，操作就超时了。

现在再看另外一个示例，它检查一组网站的请求状态码。这个脚本需要写几行代码以遍历一个网站列表，收集网站相应的状态码：

```
import urllib2

sites = [
```

```
    "http://www.google.com",
    "http://www.bing.com",
    "http://stackoverflow.com",
    "http://facebook.com",
    "http://twitter.com"
]

def check_http_status(url):
    return urllib2.urlopen(url).getcode()

http_status = {}
for url in sites:
    http_status[url] = check_http_status(url)

for url in http_status:
    print "%s: %s" % (url, http_status[url])
```

如果用Linux的时间命令行工具运行代码，就可以获得程序运行的时间：

```
$time python non_threading_httpstatus.py
```

输出结果如下所示：

```
http://www.google.com: 200
http://facebook.com: 200
http://stackoverflow.com: 200
http://www.bing.com: 200
http://twitter.com: 200

real    0m3.936s
user    0m0.060s
sys     0m0.018s
```


现在，检查代码看看我们发现了什么问题，我们显然可以把IO操作函数（check_http_status）转变为一个线程来优化代码。我们可以并发地检查所有网站的状态，不需要在一个检查完成之后再运行另一个检查：

```
import urllib2
import threading

sites = [
    "http://www.google.com",
    "http://www.bing.com",
    "http://stackoverflow.com",
    "http://facebook.com",
    "http://twitter.com"
]

class HTTPStatusChecker(threading.Thread):

    def __init__(self, url):
        threading.Thread.__init__(self)
        self.url = url
```

```
            self.status = None

        def getURL(self):
            return self.url

        def getStatus(self):
            return self.status

        def run(self):
            self.status = urllib2.urlopen(self.url).getcode()

    threads = []
    for url in sites:
        t = HTTPStatusChecker(url)
        t.start()  # 启动线程
        threads.append(t)


    # 让主线程join其他子线程,
    # 这样我们就可以在子线程完成后打印全部的结果
    for t in threads:
        t.join()

    for t in threads:
        print "%s: %s" % (t.url, t.status)
```

同样使用time命令运行新脚本：

```
$time python threading_httpstatus.py
```

我们将获得下面的输出结果：

```
http://www.google.com: 200
http://www.bing.com: 200
http://stackoverflow.com: 200
http://facebook.com: 200
http://twitter.com: 200

real    0m1.576s
user    0m0.068s
sys     0m0.016s
```

显然，线程版的程序更快。从图中对比发现，它的速度几乎是上一版的三倍，性能改善十分显著。

通过Event对象实现线程间通信

虽然线程通常是作为独立运行或并行的任务，但是有时也会出现线程间通信的需求。

threading模块提供了事件（event）对象实现线程间通信（https://docs.python.org/2/library/threading.html#event-objects）。它包含一个内部标记（internal flag），以及可以使用set()或clear()方法的调用线程（caller thread）。

Event类的接口很简单，它支持的方法如下。

- is_set：如果事件设置了内部标记，就返回True。
- set：把内部标记设置为True。它可以唤醒等待被设置标记的所有线程。调用wait()方法的线程将不再被阻塞。
- clear：重置内部标记。调用wait()方法的线程，在调用set()方法之前都将被阻塞。
- wait：在事件的内部标记被设置好之前，使用这个方法会一直阻塞线程调用。这个方法支持一个可选参数，作为等待时限（timeout）。如果等待时限非0，则线程会在时限内被一直阻塞。

让我们用线程事件对象来演示一个简单的线程通信示例，它们可以轮流打印字符串。两个线程将共享同一个事件对象。在while循环中，每次循环时，一个线程设置标记，另一个线程重置标记。每一次动作（set和clear），它们都会打印正确的字符：

```
import threading
import time


class ThreadA(threading.Thread):

    def __init__(self, event):
        threading.Thread.__init__(self)
        self.event = event

    def run(self):
        count = 0
        while count < 5:
            time.sleep(1)
            if self.event.is_set():
                print "A"
                self.event.clear()
            count += 1


class ThreadB(threading.Thread):

    def __init__(self, evnt):
        threading.Thread.__init__(self)
        self.event = evnt

    def run(self):
        count = 0
        while count < 5:
            time.sleep(1)
            if not self.event.is_set():
                print "B"
                self.event.set()
            count += 1
```

```
event = threading.Event()

ta = ThreadA(event)
tb = ThreadB(event)

ta.start()
tb.start()
```

总的来说，下表内容可以揭示Python多线程的使用时机。

使用线程	不用线程
频繁的IO操作程序	大量CPU操作程序
并行任务可以通过并发解决	程序必须利用多核心操作系统
GUI开发	

5.4　多进程

如前所述，由于GIL的存在，Python的多线程并没有实现真正的并行。因此，一些问题使用`threading`模块并不能真正解决。

不过Python为多线程提供了一个替代方案：多进程。在多进程里，线程被换成了一个个子进程。每个进程都运作各自的GIL（这样Python就可以并行开启多个进程，没有数量限制）。

需要明确的是，线程都是同一个进程的组成部分，它们共享同一块内存、存储空间和计算资源。而进程却不会与生成它们的父进程共享内存，因此进程间的通信比线程间通信更加复杂。

多进程相比多线程的优缺点如下表所示。

优　势	劣　势
可以使用多核操作系统	更多的内存消耗
进程使用独立的内存空间，避免竞态问题	进程间的数据共享变得更加困难
子进程容易中断（killable）	IPC（Interprocess communication，进程间通信）处理比线程困难
避开GIL的限制（虽然只是在CPython里才存在的问题）	

Python 多进程

`multiprocessing`模块（https://docs.python.org/2/library/multiprocessing.html）提供了一个`Process`类，它其实相当于多线程模块中的`threading.Thread`类。因此，把多线程代码迁移到多进程还是比较简单的，因为代码的基本结构是不变的。

让我们快速演示一个多进程的示例：

```
#!/usr/bin/python
```

```
import multiprocessing

def run(pname):
    print pname

for i in range(10):
    p = multiprocessing.Process(target=run, args=("Process-" + str(i), ))
    p.start()
    p.join()
```

上面的代码很简单，但是从中可以看出多进程和多线程代码非常像。



由于Windows生成子进程的机制与Linux不同，所以如果在Windows系统上运行上面的代码，请把multiprocessing放到if __name__ == '__main__':下面。代码如下所示：

```
#!/usr/bin/python
import multiprocessing

def run(pname):
    print pname

if __name__ == '__main__':

    for i in range(10):
        p = multiprocessing.Process(target=run,
        args=("Process-" + str(i), ))
        p.start()
        p.join()
```


1. 进程退出状态

当进程结束（或中断）的时候，会产生一个退出码（exitcode），它是一个数字，表示执行的结果。不同的数字分别表示进程正常完结，异常完结，或是由另一个进程中断的状态。

具体有以下三种情况：

- 等于0表示正常完结
- 大于0表示异常完结
- 小于0表示进程被另一个进程通过-1*exit_code信号终结

下面的代码用于演示如何读取和使用退出码，具体情况还要具体分析。

```
import multiprocessing
import time


def first():
    print "There is no problem here"
```

```
def second():
    raise RuntimeError("Error raised!")


def third():
    time.sleep(3)
    print "This process will be terminated"

workers = [multiprocessing.Process(target=first), multiprocessing.Process(
    target=second), multiprocessing.Process(target=third)]

for w in workers:
    w.start()

workers[-1].terminate()

for w in workers:
    w.join()


for w in workers:
    print w.exitcode
```

代码输出结果如下图所示。

```
There is no problem here
Process Process-2:
Traceback (most recent call last):
  File "/usr/lib/python2.7/multiprocessing/process.py", line 258, in _bootstrap
    self.run()
  File "/usr/lib/python2.7/multiprocessing/process.py", line 114, in run
    self._target(*self._
  File "multiprocessing_                    , line 10, in second
    raise Runtime    ror("Error raised!")
RuntimeErr  : Error raised!
0
1
-15
```

退出代码列表

注意第三个子进程的print语句没有执行。这是因为在sleep方法结束之前进程已经被中止了。还有一点需要注意的是，两个独立的for循环处理三个子进程：一个启动进程，另一个通过join方法连接进程。如果我们在开启每个子进程时都执行join()方法（而不是没使用join()就把第三个进程中断），那么第三个进程就不会失败。于是第三个子进程返回的退出码也将是0，因为和多线程一样，join()方法在目标进程完结之前会阻塞子进程的调用。

2. 进程池

多进程模块还提供了Pool类（https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing.pool），表示一个进程池，里面装有子进程，可以通过不同的方法执行一组任务。

Pool类的主要方法如下。

- apply：这个方法在独立的子进程中运行一个函数。它还会在被调用函数返回结果之前阻

塞进程。

- apply_async：这个方法会在独立子进程中异步地运行一个函数，就是说进程会立即返回一个ApplyResult对象。要获得真实的返回值，需要使用get()方法。get()在异步执行的函数结束之前都会被阻塞。
- map：这个方法对一组数值应用同一个函数。它是一个阻塞动作，所以返回值是每个值经过函数映射的列表。

上面的方法提供了不同的方式遍历你的数据，可以是异步、同步，也可以是逐个处理。具体方法根据需求进行选择。

3. 进程间通信

之前已经提过，进程间通信的方式不像线程间通信那么简单。但是，Python提供了一些工具帮助我们解决问题。

Queue类是一个既线程安全又进程安全的先进先出（FIFO，first in first out，https://docs.python.org/2/library/multiprocessing.html#exchanging-objects-between-processes）数据交换机制。multiprocessing提供的Queue类基本是Queue.Queue的克隆版本，因此两者API基本相同。下面的代码是演示两个进程利用Queue进行通信的示例：

```
from multiprocessing import Queue, Process
import random


def generate(q):
    while True:
        value = random.randrange(10)
        q.put(value)
        print "Value added to queue: %s" % (value)


def reader(q):
    while True:
        value = q.get()
        print "Value from queue: %s" % (value)


queue = Queue()
p1 = Process(target=generate, args=(queue,))
p2 = Process(target=reader, args=(queue,))

p1.start()
p2.start()
```

5

(1) Pipe方法

Pipe（管道）方法（https://docs.python.org/2/library/multiprocessing.html#exchanging-objects-

between-processes）为两个进程提供了一种双向通信的机制。`Pipe()`函数返回一对连接对象，每个对象表示管道的一端。每个连接对象都有`send()`和`recv()`方法。

下面的代码表示一个简单的`Pipe`用法,与前面的`Queue`示例类似。这个脚本会创建两个进程：一个进程产生随机数通过管道发送出去，另一个进程接收数据，然后写入文件中：

```
from multiprocessing import Pipe, Process
import random


def generate(pipe):
    while True:
        value = random.randrange(10)
        pipe.send(value)
        print "Value sent: %s" % (value)


def reader(pipe):
    f = open("output.txt", "w")
    while True:
        value = pipe.recv()
        f.write(str(value))
        print "."


input_p, output_p = Pipe()
p1 = Process(target=generate, args=(input_p,))
p2 = Process(target=reader, args=(output_p,))

p1.start()
p2.start()
```

(2) `Event`

多进程中也有事件`Event`，它们的工作方式与多线程类似。开发者唯一需要记住的是，事件对象不能被传递到子进程函数中。如果你那么做，就会导致运行时错误，信号[1]对象只能通过继承机制在进程间共享。也就是说，你不能写如下所示的代码：

```
from multiprocessing import Process, Event, Pool
import time


event = Event()
event.set()


def worker(i, e):
    if e.is_set():
```

① semaphore，操作系统概念。——译者注

```
        time.sleep(0.1)
        print "A - %s" % (time.time())
        e.clear()
    else:
        time.sleep(0.1)
        print "B - %s" % (time.time())
        e.set()


pool = Pool(3)
pool.map(worker, [(x, event) for x in range(9)])
```

而是应该这样写：

```
from multiprocessing import Process, Event, Pool
import time


event = Event()
event.set()


def worker(i):
    if event.is_set():
        time.sleep(0.1)
        print "A - %s" % (time.time())
        event.clear()
    else:
        time.sleep(0.1)
        print "B - %s" % (time.time())
        event.set()


pool = Pool(3)
pool.map(worker, range(9))
```

5.5　小结

本章介绍了两种多任务处理方式（多进程，多线程），以及各自的特性和优缺点，而具体如何选择完全由开发者自行决定。由于它们适用于不同的场景，所以并非一个绝对比另一个好，虽然它们看起来像是在解决同样的问题。

这一章需要掌握的重点是之前提到的两点：两种方法的主要特性，以及两种方法的使用时机。

下一章将继续介绍优化工具。这一次，我们将介绍Cython（一种编译器，可以把Python代码编译成C语言）和PyPy（一种Python实现的解释器，没有CPython的GIL）。

第 6 章

常用的优化方法

6

在学习性能优化的路上，我们从第4章开始，首先介绍了一些优化方法，然后在第5章中深入研究了两种重要的优化策略：多线程和多进程。我们还对两种策略的用法以及使用场景进行了详细的介绍。

归根到底，我们都是在优化Python众多实现中的一种（CPython）。然而还有其他实现方式，在这一章我们将介绍其中的两种。

- 我们将介绍PyPy，它是Python解释器的另一个版本，本书一直在使用。相比标准版Python，它有一些优势。
- 我们还会介绍Cython，一种优化过的静态编译器，可以让我们写静态代码，并轻松借助C和C++的力量。

两种方法都可以让开发者以更高效的方式运行代码，当然也得考虑代码的具体特点。对于每种方法，我们都会详细介绍其特性、安装方法，以及使用它们编写的代码示例。

6.1 PyPy

和CPython是用C语言写成的Python的标准实现一样，PyPy是Python的另一种实现，有2.x版本和3.x版本。它用RPython模拟语言的功能，RPython是一种静态类型的Python版本。

PyPy项目（http://pypy.org/）是另一个旧项目Psycho的延续，Psycho是一种用C语言写的Python的JIT编译器。它在32位英特尔处理器上运行得很不错，但是一直没有更新。其最新稳定版还是在2007年发布的，现在已经废弃了。PyPy在2007年发布了1.0版。虽然它一开始只算是一个研究性的项目，但是持续了很多年，最终在2010年发布了1.4版。这一版发布之后，用PyPy系统可以开发正式的产品，并且可以与Python 2.5版兼容，这令人们对PyPy的信心不断增强。

PyPy最新的稳定版是2014年6月发布的2.5版，与Python 2.7兼容。同时还发布了PyPy3测试版本，可以与Python 3.x版本兼容。

我们选择PyPy作为优化脚本的可靠方法，理由如下。

- 速度：PyPy的一个主要特性是对普通Python代码运行速度的优化。这是由于它使用JIT（Just-in-time）编译器。在静态编译代码时它提供了一种灵活性，可以在运行时根据运行环境（处理器、操作系统版本等）进行调整。另一方面，静态编译程序可能需要一个可执行或者不同条件的组合体。
- 内存：PyPy执行脚本时消耗的内存要比普通Python小。
- 沙盒（sandboxing）：PyPy提供了沙盒环境，在调用C语言库的时候使用。这种机制会与一个处理实际情况的外部进程通信。虽然这种机制很好，但还只是一个原型，需要一些处理才能正常使用。
- 无栈（stackless）：PyPy还提供了与Stackless Python（http://www.stackless.com/）相似的特性。有人甚至认为PyPy比后者更加强大和灵活。

6.1.1　安装 PyPy

有一些方法可以安装PyPy。

- 你可以直接从网页上（http://pypy.org/download.html#default-with-a-jit-compiler）下载可执行文件。要下载正确的版本，请根据页面下载链接的操作系统标识对号入座。如果下载的系统标识没对上，很可能就装不上了。

may have more luck trying out Squeaky's portable Linux binaries.

请根据自己的操作系统选择版本

Python2.7 compatible PyPy 2.5.1

- Linux x86 binary (32bit, tar.bz2 built on Ubuntu 12.04 - 14.04) (see [1] below)
- Linux x86-64 binary (64bit, tar.bz2 built on Ubuntu 12.04 - 14.04) (see [1] below)
- ARM Hardfloat Linux binary (ARMHF/gnueabihf, tar.bz2, Raspbian) (see [1] below)
- ARM Hardfloat Linux binary (ARMHF/gnueabihf, tar.bz2, Ubuntu Raring) (see [1] below)
- ARM Softfloat Linux binary (ARMEL/gnueabi, tar.bz2, Ubuntu Precise) (see [1] below)
- Mac OS/X binary (64bit)
- Windows binary (32bit) (you might need the VS 2008 runtime library installer vcredist_x86.exe.)
- Source (tar.bz2); Source (zip). See below for more about the sources.
- All our downloads, including previous versions. We also have a mirror, but please use only if you have troubles accessing the links above

如果你使用的是Linux发行版或OS X系统，可以先看看系统的安装包仓库里是否有PyPy安装包。通常，很多系统都有PyPy包，比如Ubuntu、Debian、Homebrew、MacPorts、Fedora、

Gentoo、Arch。如果你用的是Ubuntu系统，可以通过下面的命令安装：

```
$ sudo apt-get install pypy
```

- 最后一种方法是下载源代码，然后自己编译。这可能比直接下载可执行文件安装要麻烦。但是，如果操作正确，就可以保证你安装的PyPy完全与你的系统兼容。



事先提个醒，从源代码编译听起来像是很简单的事情，但是真正做起来是很花时间的。在一台i7、8G内存的电脑上，编译过程需要消耗1个小时左右。编译内容如下图所示。

```
[Timer] Timings:
[Timer] annotate                         ---   383.3 s
[Timer] rtype_lltype                     ---   558.3 s
[Timer] pyjitpl_lltype                   ---   666.0 s
[Timer] backendopt_lltype                ---   172.5 s
[Timer] stackcheckinsertion_lltype       ---   221.3 s
[Timer] database_c                       ---   216.2 s
[Timer] source_c                         ---   275.1 s
[Timer] compile_c                        ---   956.9 s
[Timer] ==========================================
[Timer] Total:                           --- 3449.5 s
```

6.1.2　JIT 编译器

这是PyPy的主要特性之一，是PyPy可以在运行速度上远胜普通Python（CPython）的关键。

根据PyPy官方网站提供的性能测试数据，虽然不同的任务会有差异，但是一般情况下PyPy编译器的速度要比CPython快7倍。

通常，普通版Python的编译器在程序第一次运行之前，要把全部源代码都转换成机器码。但是，我们可以不这么做。这是标准编译器的处理步骤（预处理并转换源代码，然后组合并链接库函数）。

JIT编译是指源代码编译是在运行时同时进行的，而不像标准编译器那样在运行前进行。代码的处理方式分成两步。

(1) 首先，源代码被翻译成一种中间语言代码。在一些编程语言中，比如Java里，称为字节码。
(2) 有了字节码之后，我们开始把它编译并翻译成机器码，但是按需翻译。JIT编译器的特性之一就是，只编译需要运行的那部分代码，不是一次性全编译。

第二步就是JIT编译器与其他解释型语言（如CPython）在字节码被解释而不是被编译时的根本差异。另外，JIT编译器还会缓存已编译的代码，这样在下一次编译时可以避免多余的消耗。

了解这些特性之后，很明显，程序如果要利用JIT编译器，就必须运行几秒钟，以便指令缓存起作用。但是，实际效果可能与期待的相反，因为编译消耗的时间是开发者真正会注意到的唯

一时间差异。

使用JIT编译器的一个主要优势是，被执行的程序可以在具体的操作系统上优化机器码（包括CPU、操作系统等）。因此它实现了一种普通静态编译程序（甚至解释型程序）无法获得的灵活性。

6.1.3 沙盒

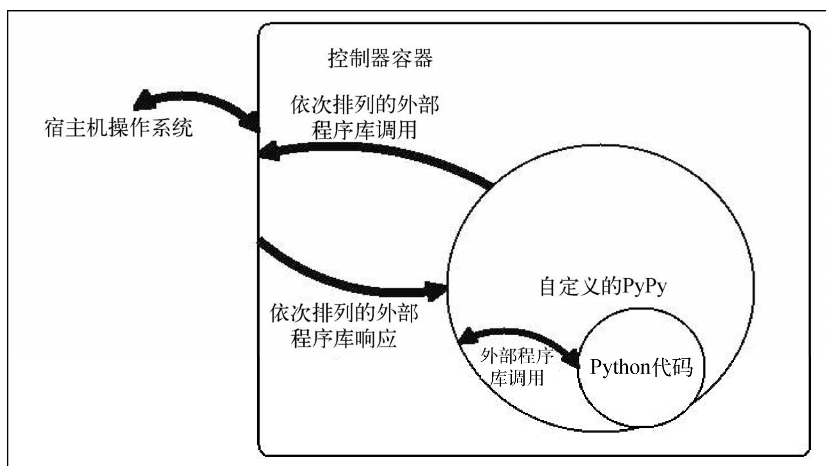
虽然PyPy的沙盒模型仍然被认为只是一个原型，但是我们简要介绍其工作原理，以帮助读者理解其特性。

沙盒可以理解成是一个安全运行环境，不安全的Python代码也可以在其中运行，不必担心其会损坏整个宿主系统。

PyPy的沙盒通过一个双进程模型实现。

(1) 一方面，我们有一个自定义的PyPy专门编译沙盒模型中的函数。也就是说任何库或系统调用（例如I/O操作），都会在一个`stdout`里排列好等待响应。

(2) 另一方面，我们有一个容器进程，可以用PyPy或CPython运行。这个进程主要是响应PyPy内部进行的库和系统调用。



上图显示的是，一段Python代码在沙盒模式中运行，并执行一个外部库调用的过程。

容器进程就是决定使用哪种虚拟化的进程。例如，创建文件句柄的内部进程，其实是容器进程伪装的。这个进程是沙盒进程与操作系统的中间层。

需要注意的是，前面介绍的沙盒运行机制和语言层面的沙盒并不相同。开发者可以使用整个

6

指令集。因此，你可以用代码实现一个透明、安全的系统，可以运行在标准、安全的系统上。

6.1.4　JIT 优化

我们之前介绍过，JIT使得PyPy区别于CPython的实现。同样的JIT特性也可以让Python快速运行。

直接使用PyPy运行Python代码，我们就可以获得更快的速度。但是，如果我们想更深入地优化代码，就需要考虑一些细节。

1. 针对函数的优化

JIT可以分析函数热度，即判断哪个函数比其他函数“更热”（hotter，执行次数更多）。因此，我们需要更合理地构造函数代码，尤其是那些需要频繁调用的函数。

让我们快速浏览一个例子。下面的代码对比了两种函数的使用方式，一种是将代码直接内联在其他函数中，另一种是把代码封装成一个单独的函数，在其他函数内调用。后者消耗的时间就会包括函数查询和函数调用两部分时间：

```
import math
import time

TIMES = 10000000

init = time.clock()
for i in range(TIMES):
    value = math.sqrt(i * math.fabs(math.sin(i - math.cos(i))))

print "No function: %s" % ( init - time.clock())

def calcMath(i):
    return math.sqrt(i * math.fabs(math.sin(i - math.cos(i))))
init = time.clock()

for i in range(TIMES):
    value = calcMath(i)
print "Function: %s" % ( init - time.clock())
```

这段代码很简单，但是你会发现用PyPy的第二种方法更快。普通的CPython则结果相反，因为没有对代码进行及时优化。第二种方法因为函数查询和函数调用造成了额外的消耗。但是，PyPy和它的JIT又一次证明，如果你想按照它们的方式优化代码，就要放弃旧理念。

```
fernando@dune:~/workspace/writing/python/chapter6$ pypy pypy_jit_test.py
No function: -0.910388
Function: -0.909176
fernando@dune:~/workspace/writing/python/chapter6$ python pypy_jit_test.py
No function: -4.332882
Function: -4.899006
```

JIT优化代码

CPython标准代码

通过上图可以验证两条我们一直在强调的结论。

- 运行同样的代码时，PyPy比CPython要快。
- JIT会实时优化代码，而CPython不会。

2. 考虑用`cStringIO`连接字符串

这不是一个小优化，可同时适用于两种代码优化。我们已经介绍过，在Python里字符串是不可变对象。因此，如果我们想把大量的字符串连接成一个对象，最好是换一种数据类型，而不是用原来的字符串类型，因为用原来的字符串连接性能会很差。

在PyPy里其实也是这样。不过，我们不用列表对象，而是用`cStringIO`模块（http://pymotw.com/2/StringIO/），我们将看到通过它可以获得更好的性能。

需要注意的是，因为PyPy本质上是Python实现，所以用`cStringIO`替换`StringIO`会让人困惑，毕竟我们要用一个C语言库而不是一个纯Python库。这是正确且有效的做法，因为一些CPython常用的C语言库同样支持PyPy。根据我们的情况，用下面的例子对比连接同样字符串的三种方式（分别用普通字符串、`cStringIO`模块和列表）的消耗时间：

```
from cStringIO import StringIO
import time

TIMES = 100000

init = time.clock()
value = ''
for i in range(TIMES):
    value += str(i)
print "Concatenation: %s" % (init - time.clock())


init = time.clock()
value = StringIO()
for i in range(TIMES):
    value.write(str(i))
print "StringIO: %s" % (init - time.clock())

init = time.clock()
value = []
for i in range(TIMES):
    value.append(str(i))
finalValue = ''.join(value)
```

```
print "List: %s" % (init - time.clock())
```

通过三种方式的消耗时间可以看出，在PyPy中StringIO对象是最快的。它比普通字符串快，甚至比用列表的方法还要好。

如果我们再用CPython运行代码，就会获得不同的结果。最好的运行结果还是用列表。

```
fernando@dune:~/workspace/writing/python/chapter6$ pypy pypy_str_vs_stringio.py
Concatenation: -5.557533
StringIO: -0.005191
List: -0.009325
fernando@dune:~/workspace/writing/python/chapter6$ python pypy_str_vs_stringio.py
Concatenation: -0.028069
StringIO: -0.031069
List: -0.021833
```

PyPy 对StringIO的优化效果更好

CPython相反，列表join方法的优化效果更好

上图证实了以上结论。注意看PyPy的第一种方法的性能是多么地差。

3. 禁止JIT的操作

有一些具体的方法虽然不是直接优化的手段，但是使用它们会消除PyPy的JIT效果。因此，了解这些方法非常重要。

下面三种方法会通过sys模块禁止JIT效果（通过目前的PyPy版本；当然，以后应该会改变）。

- _getframe：这个方法会从callstack返回一个frame对象，也可以接受一个从callstack发出的带深度参数的callstack对象作为参数。这么做性能损失非常大，非万不得已最好别用，比如系统调试的时候。
- exc_info：这个方法会返回一个三元素的元组，提供待处理异常的相关信息。三个元素分别是type、value和traceback，具体解释如下。
  - type：待处理的异常类型。
  - value：异常参数。
  - traceback：跟踪traceback对象，当异常被抛出时，里面会封装一个callstack对象。
- settrace：这个方法可以设置跟踪函数。它能让你从Python内部跟踪Python代码。就像前面提到的，这个方法也是万不得已时才用，因为它在执行时会禁止JIT。

6.1.5　代码示例

作为这个主题的最后一个例子，让我们看看great_circle函数的代码（后面会介绍）。大圈（great circle）计算用于计算地球上任意两点间的距离。

这个脚本会在for循环中重复500万次。其实，脚本反复地调用同一个函数（确切地说是500

万次）。这种情况用CPython解释器并不合适，因为函数查询次数非常多。

但是，就像我们前面介绍过的，反复调用函数的代码可以利用PyPy的JIT技术进行优化。基本上可以认为，代码只要用PyPy运行就已经得到了优化：

```
import math

def great_circle(lon1, lat1, lon2, lat2):
    radius = 3956  # 英里
    x = math.pi/180.0

    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = math.acos((math.cos(a)*math.cos(b)) +
                  (math.sin(a)*math.sin(b)*math.cos(theta)))
    return radius*c

lon1, lat1, lon2, lat2 = -72.345, 34.323, -61.823, 54.826
num = 5000000
for i in range(num):
    great_circle(lon1, lat1, lon2, lat2)
```

前面的内联函数也可以按照我们之前介绍过的方法进行优化。我们可以把great_circle函数中的一行代码移出来，封装成一个函数，然后再运行代码，如下所示：

```
import math

def calcualte_acos(a, b, theta):
    return math.acos((math.cos(a)*math.cos(b)) +
                     (math.sin(a)*math.sin(b)*math.cos(theta)))

def great_circle(lon1, lat1, lon2, lat2):
    radius = 3956  # 英里
    x = math.pi/180.0

    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = calcualte_acos(a, b, theta)
    return radius*c

lon1, lat1, lon2, lat2 = -72.345, 34.323, -61.823, 54.826
num = 5000000

for i in range(num):
    great_circle(lon1, lat1, lon2, lat2)
```

你会发现我们把acos函数所在的行封装成了独立的函数，因为它是函数中最耗费时间的一行（里面一共调用了6个trig函数）。把这行代码封装成函数之后，就可以用JIT对函数调用进行优化了。

总之，当我们做了一点儿改变，并用PyPy执行代码之后，函数的运行时间是0.5秒。另一方面，如果我们在CPython里运行代码，运行时间将是4.5秒（在我自己的电脑上运行），相当地慢。

6.2 Cython

从技术角度看，Cython（http://cython.org/）并没有使用另一种与CPython不同的解释器，但是它可以让我们直接将Python代码编译成C语言（CPython不会这么做）。

你会看到Cython其实是一个转换器，可以简单看成一个软件，它可以把源代码从一种语言翻译成另一种语言。类似的软件还有CoffeeScript和Dart。这两个是不同的软件，使用不同的语言，但是都翻译成JavaScript。

Cython把Python的超集（扩展版本）翻译成C/C++。然后，它会被编译成Python模块。这样允许开发者：

- 用Python代码调用原生C/C++
- 用静态类型声明把Python代码优化成C语言的性能

静态类型是Cython这个翻译器产生优化的C语言代码的主要特征，可以把Python的动态特性转变成静态且更快的代码（有时候可以达到几个数量级）。

不过这么做会把代码变得更啰嗦，会破坏代码的可维护型和可读型。因此，通常并不推荐使用静态类型，除非有充分理由证明增加静态类型可以充分提高代码的性能。

开发者可以使用所有的C类型。Cython可以对变量赋值自动进行类型转换。当面对Python的任意长度整数时，如果转换成C类型出现了栈溢出，Python的溢出错误就会产生。

下图显示的是用纯Python与Cython编写的同样的代码。

Python	Cython

`def f(x):`
`    return x**2-x`
 	
`def integrate_f(a, b, N):`
`    s = 0`
`    dx = (b-a)/N`
`    for i in range(N):`
`        s += f(a+i*dx)`
`    return s * dx`
	`def f(double x):`
	`    return x**2-x`
	
	`def integrate_f(double a,`
	`double b, int N):`
	`    cdef int i`
	`    cdef double s, dx`
	`    s = 0`
	`    dx = (b-a)/N`
	`    for i in range(N):`
	`        s += f(a+i*dx)`
	`    return s * dx`

两种代码的主要差异用加粗字体显示。差异只是变量的类型定义，包括两个函数的参数，以及局部变量。除此之外，左边的Cython代码可以生成优化的C语言代码。

6.2.1 安装 Cython

在你的系统上安装Cython有好几种方法。但是每种方法的共同步骤是在安装之前都需要一个C语言编译器。我们不会深入介绍这个步骤，因为不同系统的安装命令不一样。

安装C编译器之后，可以用下面的步骤安装Cython。

(1) 从网站（http://cython.org）下载最新版源代码，解压文件进入目录，运行下面的命令：

```
$python setup.py install
```

(2) 如果你的系统上有安装工具（比如pip），可以用下面的命令：

```
$pip install cython
```



如果你用下面的开发环境，Cython应该已经安装好了。即使没装，你也可以用更简单的方法安装：

- Anaconda
- Enthought Canopy
- PythonXY
- Sage

6.2.2 建立一个 Cython 模块

Cython可以把代码编译成C模块，然后导入代码。要实现这个目标，你需要完成以下几个步骤。

(1) 首先，需要用Cython把.pyx文件编译（翻译）成.c文件。这些文件里的源代码，基本都是纯Python代码加上一些Cython代码。后面我们会看到一些例子。

(2) 然后，.c文件被C语言编译器编译成.so库，这个库之后可以导入Python。

(3) 编译代码有一些方法，如下所示。

- 我们可以创建一个distutils配置文件。bistutils是一个创建其他模块的工具，我们可以用它生成自定义的C语言编译文件。
- 运行cython命令将.pyx文件编译成.c文件。然后用C语言编译器把C代码手动编译成库文件。
- 最后一种方法是用pyximport，像导入.py文件一样导入.pyx直接使用。

(4) 为了演示前面介绍的知识点，我们用distutils方法看看下面的例子：

```
# test.pyx
def join_n_print(parts):
    print ' '.join(parts)

# test.py
```

6

```
from test import join_n_print
join_n_print(["This", "is", "a", "test"])

# setup.py
from distutils.core import setup
from Cython.Build import cythonize

setup(
  name='Test app',
  ext_modules=cythonize("test.pyx"),
)
```

(5) 就是这么简单！在上面的代码中，要被导出的代码在.pyx文件中。setup.py文件一直都是这样。它可以通过不同的参数调用`setup`函数。最后，它会调用test.py文件，文件中导入并使用了库文件。

(6) 为了有效地编译代码，可以用下面的命令：

```
$ python setup.py build_ext -inplace
```

上面的命令生成的结果如下图所示。你会看到它不仅翻译（Cython化）代码，而且用C语言编译器把编译代码成库文件。

```
Compiling test.pyx because it changed.
Cythonizing test.pyx
running build_ext
building 'test' extension
x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes
.7/test.o
x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,-Bsymbolic-functions -Wl,-
rototypes -D_FORTIFY_SOURCE=2 -g -fstack-protector --param=ssp-buffer-size=4 -Wformat -Werror=format-
orkspace/writing/python/chapter6/test.so
```

前面的代码是一个非常简单的模型。但是，面对复杂的情况时，Cython通常都需要导入两类文件。

- **定义文件**：文件扩展名.pxd，是其他Cython文件要使用的变量、类型、函数名称的C语言声明。
- **实现文件**：文件扩展名.pyx，包括在.pxd文件中已经定义好的函数实现。

定义文件中通常包括C类型声明、外部C函数或变量声明，以及模块中定义的C函数声明。它们不包含任何C或Python函数的实现，也不包含任何Python类的定义或可执行代码行。

另一方面，在实现文件中可以有几乎所有的Cython语句。

下面是Cython官方文档（http://docs.cython.org/src/userguide/sharing_declarations.html）里提供的一个经典的两文件模块示例，里面还演示了如何导入.pyx文件：

```
# dishes.pxd
```

```
cdef enum otherstuff:
    sausage, eggs, lettuce

cdef struct spamdish:
    int oz_of_spam
    otherstuff filler

# restaurant.pyx:
cimport dishes
from dishes cimport spamdish

cdef void prepare(spamdish * d):
    d.oz_of_spam = 42
    d.filler = dishes.sausage

def serve():
    cdef spamdish d
    prepare( & d)
    print "%d oz spam, filler no. %d" % (d.oz_of_spam, d.filler)
```

默认情况下，当运行`cimport`时，它会在搜索路径中查找同名模块的modulename.pxd文件。无论定义文件何时改变，导入的文件都需要重新编译。好在`Cythin.Build.cythonize`功能可以帮我们解决这个问题。

6.2.3 调用 C 语言函数

和标准Python一样，Cython也可以让开发者直接通过调用已编译的C函数与C语言交互。导入这些库函数的方式与标准Python类似：

```
from libc.stdlib cimport atoi
```

在定义文件或实现文件中使用`cimport`，是为了导入在其他文件中定义的名称。这个语法和标准Python中的`import`完全一致。

如果你还需要使用库文件中已定义的一些类型的定义，那么可能需要头文件（.h文件）。对于这种情况，用Cython不像在C语言中直接引用文件那么简单，还需要重新声明你将要使用的类型和结构：

```
cdef extern from "library.h":
    int library_counter;
    char *pointerVar;
```

上面的代码演示了Cython的特性。

- 它让Cython知道如何在生成的C语言代码中放入`#include`语句来引用我们需要使用的库。
- 它会防止Cython为这段代码中的声明生成任何C代码。

6

- 它会把代码中的所有声明看成`cdef extern`，表示那些声明是定义在其他地方。

值得注意的是，这个语法必须使用，因为Cython在任何时刻都不会读取头文件的内容。所以你还需要把头文件的内容进行重新声明。你只需要重新你需要的那些，不需要关心代码中的其他内容。例如，如果你在头文件中声明了一个成员很多的大结构体，你只需要重新声明你需要使用的成员。在编译时，C编译器会使用完整的结构体源代码。

解决命名冲突

当导入的函数名称与另一个函数名称相同时，会出现一个有趣的问题。

假如你在头文件myHeader.h中定义了一个函数`print_with_colors`，而你希望把它封装在同名的`print_with_colors`函数里；Cython提供了一种方法可以让你绕开这个问题，并保留你想要的名称。

你可以在Cython的声明文件（.pxd）中增加`extern`函数声明，然后再把它`cimport`到Cython代码中：

```
#my_declaration.pxd
cdef extern "myHeader.h":
    void print_with_colors(char *)

#my_cython_code.pyx
from my_declaration cimport print_with_colors as c_print_with_colors

def print_with_colors(str):
    c_print_with_colors(str)
```

你还可以不重命名函数，而用声明文件名作为前缀：

```
#my_cython_code.pyx
cimport my_declaration
def print_with_colors(str):
    my_declaration.print_with_colors(str)
```



两种方法都可以，开发者可根据自己的习惯自行选择。关于这个主题的更多信息，请参考文档http://docs.cython.org/src/userguide/external_C_code.html。

6.2.4　定义类型

就像前面提到的，Cython允许开发者自定义变量类型或函数返回值类型。这两种情况都要用关键词`cdef`。类型定义其实是可选项，因为Cython通常要把代码先转变成C语言，以实现对Python代码的优化。也就是说，在需要的地方定义静态类型一定有助于优化代码。

现在让我们看一段简单的Python代码示例，看看同样的代码如何用三种方式执行：纯Python，

没有类型的编译过的Cython，以及有类型且编译过的Cython。

代码如下所示：

Python	Cython

`def is_prime(num):`
`  for j in range(2,num):`
`    if (num % j) == 0:`
`      return False`
`  return True`
	`def is_prime(int num):`
	`  cdef int j;`
	`  for j in range(2,num):`
	`    if (num % j) == 0:`
	`      return False`
	`  return True`

由于我们把for循环的变量j声明为C整型变量，Cython将会把for循环转换成优化的C循环，这是代码主要的改进之一。

现在我们配置一个主文件来导入模块：

```
import sys
from < right-module-name > import is_prime

def main(argv):

    if (len(sys.argv) != 3):
        sys.exit('Usage: prime_numbers.py <lowest_bound> <upper_bound>')

    low = int(sys.argv[1])
    high = int(sys.argv[2])

    for i in range(low, high):
        if is_prime(i):
            print i,

if __name__ == "__main__":
    main(sys.argv[1:])
```

然后执行命令：

```
$ time python script.py 10 10000
```

我们会获得下面的有趣结果：

纯Python	已编译无类型	已编译有类型
0.792秒	0.694秒	0.043秒

尽管非优化版的Cython代码也比纯Python要快，但当我们开始声明类型之后才会看到Cython的真正威力。

6.2.5 定义函数类型

Cython中有两种不同类型的函数可以定义。

- **标准Python函数**：这种普通函数与纯Python代码中声明的函数完全一样。要定义这种函数，你只需要用标准的`cdef`关键字就行。这种函数接受Python对象作为参数，也返回Python对象。
- **C函数**：这种函数是是标准函数的优化版。它们可以用Python对象和C语言类型作为参数，返回值也可以是两种类型。要定义这种函数，你需要用特殊关键字`cdef`。

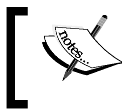
这两种函数都可以通过Cython模块调用。但是（这里有一个十分重要的差异），如果你想从Python代码中调用函数，你必须得确保函数是标准Python函数，或者是使用特殊的`cpdef`关键字。这个关键字会创建一个函数的封装对象。当用Cython调用函数时，它用C语言对象；当从Python代码中调用函数时，它用纯Python函数。

当我们要为这个函数的参数定义C语言类型时，一个自动的转换（如果可能）会将Python对象转换成C语言类型。目前可以使用的类型只有数值类型、字符串`string`和结构体`struct`类型。如果你用其他类型，后面都会产生编译时错误。

下面这段简单的代码演示了两种方式的差异：

```
#my_functions.pxd
# 这是一个纯Python函数，因此Cython会让这个函数返回并接收一个Python对象，而不是C语言原生类型。
cdef full_python_function (x):
    return x**2

# 这个函数就不同了，由于使用了cpdef关键字，所以它既是一个标准函数，也是一个优化过的C语言函数。
cpdef int c_function(int num):
    return x**2
```



如果返回值的类型或参数类型未定义，将它被看成Python对象。

还有一点需要注意的是，不返回Python对象的C语言函数，在调用时不能抛出Python异常。因此，在错误发生时，会出现警告信息，但是异常信息会被忽略。这显然是个问题。好在有一种方法可以解决这个问题。

我们可以在函数定义中使用`except`关键字。这个关键字的含义是，任何时候当函数出现异常时，都会返回一个特定的值。例子如下：

```
cdef int text(double param) except -1:
```

上面这行代码的意思是，任何时候函数一旦出现异常，就返回`-1`。关键是你不需要从函数中手动返回异常值。如果你把`False`定义成异常返回值，这么做就有意义了，因为任意`False`值都会返回。

有时候`except`返回值也可能是一个真实的返回值，这时可以用另一种表示方法：

```
cdef int text(double param) except? -1:
```

这个?表示-1可能是异常返回值。当返回-1时，Cython会调用PyErr_Occurred()判断那是一个异常还是一个正常返回值。

还有一种except表现形式，会对任意返回值调用PyErr_Occurred()函数：

```
cdef int text(double param) except *:
```

这种方法的唯一用处是诊断那些没有返回值的void函数的异常。这是因为在这种特殊的情况下，没有任何返回值可检查；但是，真正需要用这种方法的情况其实不会出现。

6.2.6 Cython 示例

让我们快速演示一个在6.1节里介绍过的例子。通过例子我们会看到如何改善脚本的性能。代码将会做同样的计算500万次，每次计算都需要使用数学库里的PI、acos、cos和sin：

```
def great_circle(lon1, lat1, lon2, lat2):
    radius = 3956  # 英里
    x = PI/180.0

    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = acos((cos(a)*cos(b)) + (sin(a)*sin(b)*cos(theta)))
    return radius*c
```

然后我们用下面的脚本来测试函数运行500万次的时间：

```
from great_circle_py import great_circle

lon1, lat1, lon2, lat2 = -72.345, 34.323, -61.823, 54.826
num = 5000000

for i in range(num):
    great_circle(lon1, lat1, lon2, lat2)
```

就像之前介绍过的，如果用Linux的time功能统计CPython解释器的运行时间，我们将会看到运行时间大概是4.5秒（在我自己的电脑上）。你的运行时间可能会不一样。

不用前面几章介绍的性能分析器，我们直接统计Cython代码。我们将直接向测试代码中引入一些在前面介绍过的优化方法。

下面是我们的第一次优化：

```
# great_circle_cy_v1.pyx
from math import pi as PI, acos, cos, sin

def great_circle(double lon1, double lat1, double lon2, double lat2):
    cdef double a, b, theta, c, x, radius

    radius = 3956  # 英里
```

```
    x = PI/180.0

    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = acos((cos(a)*cos(b)) + (sin(a)*sin(b)*cos(theta)))
    return radius*c

# great_circle_setup_v1.py
from distutils.core import setup
from Cython.Build import cythonize

setup(
    name = 'Great Circle module v1',
    ext_modules = cythonize("great_circle_cy_v1.pyx"),
)
```

你会看到，在前面的代码中，我们所做的事情就是把代码中的变量和参数都改成C语言类型。这样处理后运行时间从4.5秒降低到了3秒。我们缩短了1.5秒，但是还可以进一步优化。

现在代码中用的还是Python的`math`库。而Cython可以让Python和C语言库完美结合，这样在我们需要的时候将十分有用。你会发现，它可以满足我们的需求，而且不需要太多付出。下面让我们把Python的数学库删掉，改成C语言的math.h文件：

```
# great_circle_cy_v2.pyx
cdef extern from "math.h":
    float cosf(float theta)
    float sinf(float theta)
    float acosf(float theta)

def great_circle(double lon1, double lat1, double lon2, double lat2):
    cdef double a, b, theta, c, x, radius
    cdef double pi = 3.141592653589793

    radius = 3956  # 英里
    x = pi/180.0

    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = acosf((cosf(a)*cosf(b)) + (sinf(a)*sinf(b)*cosf(theta)))
    return radius*c
```

当我们把Python的数学库删掉，改成C语言的math.h之后，会看到前面的运行时间4.5秒变成了0.95秒，非常给力。

6.2.7 定义类型的时机选择

前面的例子可能让你觉得优化很简单。但是，对于一些较大的脚本，（尽可能地）把每个变量都转变成C语言类型，把每个Python库都替换成C语言库文件，并非最佳方案。

这么做可能会影响代码的可维护性和可读性，还可能会损害Python代码的灵活性。另外，甚至可能由于增加了大量不必要的静态类型检查和转换而损害了性能。因此必须为类型定义和库替换选择正确的对象。方法就是用Cython。Cython具有注释源代码的能力，可以图形化地显示出每行代码是如何转变成C代码的。

可以通过Cython的`-a`属性生成一个HTML文件，里面会将代码高亮显示。黄色代码行越多，表示需要换成C代码的C-API接口越多。白色代码行（没有颜色的代码行）表示已经被直接转换为C代码。让我们看看源代码在新工具下被渲染成什么样子：

```
$ cython -a great_circle_py.py
```

下图所示的HTML文件就是前面命令产生的结果。

```
Generated by Cython 0.22

Raw output: great_circle_py.c

+01: import math
 02:
+03: def great_circle(lon1,lat1,lon2,lat2):
+04:     radius = 3956 #miles
+05:     x = math.pi/180.0
 06:
+07:     a = (90.0-lat1)*(x)
+08:     b = (90.0-lat2)*(x)
+09:     theta = (lon2-lon1)*(x)
+10:     c = math.acos((math.cos(a)*math.cos(b)) +
+11:                   (math.sin(a)*math.sin(b)*math.cos(theta)))
+12:     return radius*c
```

通过上图你会清晰地看到，为了转换成C语言代码，大部分代码都需要与一些C-API接口进行交互（只有第4行是全白的）。非常重要的一点是要明白，我们的目标是要让代码行都尽可能变白。带+号的行表示代码可以点击，里面会显示产生的C代码，如下图所示。

```
Generated by Cython 0.22

Raw output: great_circle_py.c

+01: import math
 02:
+03: def great_circle(lon1,lat1,lon2,lat2):
+04:     radius = 3956 #miles
+05:     x = math.pi/180.0
    __pyx_t_1 = __Pyx_GetModuleGlobalName(__pyx_n_s_math); if (unlikely(!__pyx_t_1
    __Pyx_GOTREF(__pyx_t_1);
    __pyx_t_2 = __Pyx_PyObject_GetAttrStr(__pyx_t_1, __pyx_n_s_pi); if (unlikely(!
    __Pyx_GOTREF(__pyx_t_2);
    __Pyx_DECREF(__pyx_t_1); __pyx_t_1 = 0;
    __pyx_t_1 = __Pyx_PyNumber_Divide(__pyx_t_2, __pyx_float_180_0); if (unlikely(
    __Pyx_GOTREF(__pyx_t_1);
    __Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
    __pyx_v_x = __pyx_t_1;
    __pyx_t_1 = 0;
 06:
+07:     a = (90.0-lat1)*(x)
+08:     b = (90.0-lat2)*(x)
+09:     theta = (lon2-lon1)*(x)
+10:     c = math.acos((math.cos(a)*math.cos(b)) +
+11:                   (math.sin(a)*math.sin(b)*math.cos(theta)))
+12:     return radius*c
```

6

现在，通过对结果的判断，我们知道浅黄色表示简单的赋值（第5、7、8行和第9行）。它们很容易优化，就像我们之前做的那样：把变量都改成C语言类型，去掉原来的Python对象，而这需要我们变更代码。

代码变更完之后，分析结果如下图所示，里面是great_circle_cy_v1.pyx的分析结果。

```
Generated by Cython 0.22

Raw output: great_circle_cy_v1.c

+01: import math
 02:
+03: def great_circle(double lon1,double lat1,double lon2,double lat2):
 04:     cdef double a, b, theta, c, x, radius
 05:
+06:     radius = 3956 #miles
+07:     x = math.pi/180.0
 08:
+09:     a = (90.0-lat1)*(x)
+10:     b = (90.0-lat2)*(x)
+11:     theta = (lon2-lon1)*(x)
+12:     c = math.acos((math.cos(a)*math.cos(b)) +
+13:                   (math.sin(a)*math.sin(b)*math.cos(theta)))
+14:     return radius*c
```

现在更好了！除了第7行是黄的，其他行都是白的。当然，这是因为这一行引用了`math.pi`对象。我们可以简单地使用一个`PI`常量来初始化`pi`的值。但是，我们还有两个大黄条，第12行和第13行。这也是由于我们使用`math`库的缘故。因此，当我们去掉`math`之后，就会得到下面的文件。

```
Generated by Cython 0.22

Raw output: great_circle_cy_v2.c

 01: cdef extern from "math.h":
 02:     float cosf(float theta)
 03:     float sinf(float theta)
 04:     float acosf(float theta)
 05:
+06: def great_circle(double lon1,double lat1,double lon2,double lat2):
 07:     cdef double a, b, theta, c, x, radius
+08:     cdef double pi = 3.141592653589793
 09:
+10:     radius = 3956 #miles
+11:     x = pi/180.0
 12:
+13:     a = (90.0-lat1)*(x)
+14:     b = (90.0-lat2)*(x)
+15:     theta = (lon2-lon1)*(x)
+16:     c = acosf((cosf(a)*cosf(b)) +
 17:               (sinf(a)*sinf(b)*cosf(theta)))
+18:     return radius*c
```

上图显示的是之前演示过的代码。几乎所有的代码都可以直接翻译成C语言，我们也因此获得了很高的性能。现在我们还剩两行黄色代码：第6行和第18行。

我们没有对第6行做任何处理是因为这行是要执行的Python函数声明。如果我们用`cdef`声明函数，可能就无法接入了。但是，第18行也不完全是白的。这是因为`great_circle`是一个Python函数，其返回值是一个Python对象，需要被翻译并封装成C语言类型的值。如果我们单击这行代码，就会看到下面的结果。

```
+16:     c = acosf((cosf(a)*cosf(b)) +
 17:                (sinf(a)*sinf(b)*cosf(theta)))
+18:     return radius*c
    __Pyx_XDECREF(__pyx_r);
    __pyx_t_1 = PyFloat_FromDouble((__pyx_v_radius * __pyx_v_c)); if (
    __Pyx_GOTREF(__pyx_t_1);
    __pyx_r = __pyx_t_1;
    __pyx_t_1 = 0;
    goto __pyx_L0;
```

解决这个bug的方法是用`cpdef`声明函数，这样就可以对函数进行封装了。但是它还可以让我们声明返回值类型。所以，我们不再返回一个Python对象，而是返回一个`double`对象。最终结果如下图所示。

```
Generated by Cython 0.22

Raw output: great_circle_cy_v3.c

 01: cdef extern from "math.h":
 02:     float cosf(float theta)
 03:     float sinf(float theta)
 04:     float acosf(float theta)
 05:
+06: cpdef double great_circle(double lon1,double lat1,double lon2,double lat2):
 07:     cdef double a, b, theta, c, x, radius
+08:     cdef double pi = 3.141592653589793
 09:
+10:     radius = 3956 #miles
+11:     x = pi/180.0
 12:
+13:     a = (90.0-lat1)*(x)
+14:     b = (90.0-lat2)*(x)
+15:     theta = (lon2-lon1)*(x)
+16:     c = acosf((cosf(a)*cosf(b)) +
 17:                (sinf(a)*sinf(b)*cosf(theta)))
+18:     return radius*c
    __pyx_r = (__pyx_v_radius * __pyx_v_c);
    goto __pyx_L0;
```

通过最后一次优化，我们可以看到返回语句的C代码是如何被优化的。性能也获得了一点儿提升，执行时间从0.95秒降低到了0.8秒。

通过分析代码，我们可以进一步对代码进行细致的优化。在对Cython代码进行优化时，这种技术是展示优化进度的好方法。这种技术为代码优化的复杂性提供了可视、简单的衡量指标。



需要注意的是，有时通过Cython对代码进行优化的结果，不一定比我们在本章前半部分用PyPy对代码进行优化的效果更好（Cython的优化结果是0.8秒，而PyPy的优化结果是0.5秒）。

6.2.8 限制条件

到目前为止，我们介绍的内容都在告诉你Cython是性能优化的利器。但是，Cython的语法与Python并不完全兼容。当我们决定用这个工具优化代码之前，有一些限制条件必须要考虑。从项目公开的bug列表中，我们可以看到如下限制条件。

1. 生成器表达式

这类表达式是目前Cython最受诟病的限制之一，因为在当前的Cython中有一些问题。这些问题如下。

- 由于表达式计算范围（evaluation scope）的限定有问题，因此不能在生成器表达式内部使用可迭代对象（iterable）。
- 另外，在处理生成器表达式内部使用可迭代对象时，Cython会在生成器内部计算可迭代对象。而CPython是在生成器外部计算。
- CPython的生成器具有一些属性可以让用户查看。但是Cython的生成器的这类属性还不够全面。

2. 对比`char*`常量

目前Cython的字节字符串比较是通过指针实现的，并不是字符串的真实值：

```
cdef char* str = "test string"
print str == b"test string"
```

上面的代码并不一定会返回`True`。这将由存储第一个字符串的指针地址决定，而不是字符串的真实内容。

3. 元组作为函数参数

Python语言允许下面的语法，虽然只是一个Python 2的特性：

```
def myFunction( (a,b) ):
    return a + b
args = (1,2)
print myFunction(args)
```

但是，前面的代码在Cython中是不支持的。可能Cython以后也不会修复这个bug，因为Python 3已经不支持这个特性了。



其实Cython团队在发布1.0版本的时候就一直期望消除大多数限制。

4. 栈帧

Cython目前通过`except`返回值作为异常捕捉机制的一部分。这种方法不能捕捉`locals`和`co_code`值的异常。为了解决这类问题，需要在函数调用时生成栈帧（stack frame），因此也造成了性能损失。目前还不确定Cython团队是否会解决这个问题。

6.3 如何选择正确的工具

到此为止，我们已经介绍了两种快速优化代码的工具。但是，如何判断哪种工具是正确的呢？又或者哪种工具最好呢？

以上两个问题的答案只有一个：没有最好或正确的工具。工具无论优劣，完全由以下需求决定：

- 你需要优化的实际情况
- 对Python和C语言的熟悉程度
- 被优化代码可读性的重要程度
- 完成优化需要花费的时间

6.3.1 什么时候用 Cython

满足以下条件时，可以选择Cython。

- **你熟悉C语言**：这并不是说你得用C语言写代码，而是说你理解C语言的常用原则，比如静态类型和C语言库，如math.h头文件。因此，熟悉C语言及其原理将大有好处。
- **失去代码可读性不成问题**：用Cython写代码和Python不同，所以代码的可读性会受损。
- **需要完全支持Python的特性**：Cython虽然不是Python，但它更多地是作为Python的扩展，而不是Python的子集。因此，如果你需要完全的兼容性，Cython可以满足你的需求。

6

6.3.2 什么时候用 PyPy

满足以下条件时，可以选择PyPy。

- **你的脚本经常需要运行**：如果你的程序需要长期运行，那么PyPy的JIT优化将是一个给力的解决方案，循环运行不断优化代码。如果你的脚本只运行一次就不再用了，那么PyPy其实比普通的CPython还要慢。
- **不需要完全支持所有第三方库**：虽然PyPy和Python 2.7兼容，但是并不完全支持所有的第

三方库，尤其是那些C语言相关的库。因此，根据你的具体需求，PyPy可能不是正确选择。

- **你需要代码与CPython完全兼容**：如果你需要代码在两种环境下运行（PyPy和CPython），那么Cython肯定是没法儿满足需求了，PyPy将成为唯一选择。

6.4 小结

这一章介绍了标准Python实现的其他两个版本。一种是PyPy，它是Python的一个分支，是由RPython实现的。PyPy的JIT编译器可以在运行过程中优化代码。另一种是Cython，基本可以看成把Python代码翻译成C语言代码的转换器。我们介绍了两种方法的工作原理、安装方法，以及如何调整代码以实现性能的改善。

最后，我们总结了一些条件，以帮助读者正确地选择工具。

下一章将关注Python的具体应用领域：数据处理。这个话题在Python社区非常热门，因为Python经常被用于科学研究。我们将介绍三种工具来帮助我们改善代码性能：Numba、Parakeet和pandas。

第 7 章

用Numba、Parakeet和pandas实现极速数据处理

数据处理（number crunching）是编程世界的一个主题。但是，由于Python经常用于解决科学研究和数据科学问题，所以数据处理成了Python世界的主流课题。

通过前面六章内容的学习，我们应该可以轻松地实现自己的算法，并写出非常快速且令人满意的代码了。这六章都是针对一般的需求进行优化，而实际工作中还会有一些特殊情况需要优化。

这一章将介绍三种方法，帮助我们写出快速高效的代码来解决科学计算问题。对于每一种方法，我们都从安装开始讲起，然后通过一些示例代码体现方法的优势。

本章将介绍的三种工具如下。

- Numba：这个模块可以让你利用机器码实现高性能的纯Python代码。
- Parakeet：这是一种用Python子集为科学计算设计的运行时编译器。它非常适合处理科学计算问题。
- pandas：这个库提供了一系列高性能的数据结构和分析工具。

7.1 Numba

Numba（http://numba.pydata.org/）是一个模块，让你能够（通过装饰器）控制Python解释器把函数转变成机器码。因此，Numba实现了与C和Cython同样的性能，但是不需要用新的解释器或者写C代码。

这个模块可以按需生成优化的机器码，甚至可以编译成CPU或GPU可执行代码。

下面的代码是官方网站上的示例，可以显示模块的使用方法。我们将在后面详细介绍：

```
from numba import jit
from numpy import arange
```

```
# jit装饰器告诉Numba编译函数
# 当函数被调用时，Numba会把参数类型引入
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result

a = arange(9).reshape(3, 3)
print(sum2d(a))
```

虽然Numba看起来好像非常给力，但是它只是针对数组操作进行优化。它非常适合配合NumPy使用（我们后面会介绍）。因此，并非每个函数都可以用Numba优化，滥用Numba甚至会损害性能。

例如，让我们看一个类似的例子，不用Numba也可以完成类似的任务：

```
from numba import jit
from numpy import arange

# jit装饰器告诉Numba编译函数
# 当函数被调用时，Numba会把参数类型引入
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result

a = arange(9).reshape(3, 3)
print(sum2d(a))
```

前面的代码使用和不使用`@jit`行的效果如下。

- 使用`@jit`行：0.3秒
- 不使用`@jit`行：0.1秒

7.1.1　安装

安装Numba有两种方法：可以用Anaconda出品的`conda`包管理器，也可以复制GitHub项目源代码进行编译。

如果你打算用`conda`方法安装，需要先安装`miniconda`命令行工具（可以从http://conda.pydata.org/miniconda.html下载）。安装之后，输入下面的命令：

```
$ conda install numba
```

命令输出结果如下图所示。所有要安装、升级的包都会显示出来，像numpy和llvmlite都是与Numba有直接依赖的包。

```
fernando@dune:~$ conda install numba
Fetching package metadata: ....
Solving package specifications: .
Package plan for installation in environment /home/fernando/miniconda:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    enum34-1.0.4               |           py27_0          48 KB
    funcsigs-0.4               |           py27_0          19 KB
    llvmlite-0.4.0             |           py27_0         7.3 MB
    numpy-1.9.2                |           py27_0         7.8 MB
    numba-0.18.2               |       np19py27_1         1.1 MB
    requests-2.7.0             |           py27_0         594 KB
    setuptools-16.0            |           py27_0         341 KB
    conda-3.12.0               |           py27_0         167 KB
    pip-6.1.1                  |           py27_0         1.4 MB
    ------------------------------------------------------------
                                           Total:        18.7 MB

The following NEW packages will be INSTALLED:

    enum34:     1.0.4-py27_0
    funcsigs:   0.4-py27_0
    llvmlite:   0.4.0-py27_0
    numba:      0.18.2-np19py27_1
    numpy:      1.9.2-py27_0
    pip:        6.1.1-py27_0
    setuptools: 16.0-py27_0

The following packages will be UPDATED:

    conda:      3.10.1-py27_0 --> 3.12.0-py27_0
    requests:   2.6.0-py27_0  --> 2.7.0-py27_0

Proceed ([y]/n)?
```

另外，如果想用源代码安装，你需要先用下面的命令复制源代码：

```
$ git clone git://github.com/numba/numba.git
```

当然numpy和llvmlite包也是需要提前安装好的。都准备好之后，用下面的命令进行安装：

```
$ python setup.py build_ext -inplace
```



需要注意的是，即使没有安装依赖包，上面的命令也可以成功运行。但是如果你没有安装依赖，Numba是没法儿用的。

要检查Numba是否可以正常使用，可以在Python的REPL里输入下面的命令：

```
>>> import numba
>>> numba.__version__
'0.18.2'
```

7.1.2　使用 Numba

现在Numba已经安装好了，让我们看看如何使用它。这个模块提供的主要功能如下：

- 即时代码生成（On-the-fly code generation）
- CPU和GPU原生代码生成
- 与具有NumPy依赖的Python科学计算软件配合使用

1. Numba代码生成

Numba代码生成的主要方式是使用`@jit`装饰器。加上它就表示要用Numba的JIT编译器对函数进行优化。

在前一章里我们已经介绍过JIT编译器的好处，因此这里不再深入细节。让我们看看如何用`@jit`装饰器进行优化。

使用这个装饰器的方式有几种。默认的，也是官方推荐的方法，之前也已经介绍过：

延迟编译（Lazy compilation）

在下面的代码中，当函数被调用时，Numba将生成优化代码。它将引用属性类型和函数的返回类型：

```
from numba import jit

@jit
def sum2(a,b):
    return a + b
```

如果你用同样的函数调用其他类型，会生成并优化不同的代码路径。

(1) 及时编译

另一方面，如果你知道函数的接收类型（返回类型也可以），可以把这些类型传到`@jit`装饰器。之后，只有这种特殊情况会被优化。

下面代码中增加的部分会被传递到函数的签名里：

```
from numba import jit, int32

@jit(int32(int32, int32))
def sum2(a,b):
    return a + b
```

用于指定函数签名的常用类型如下。

- `void`：函数返回值类型，表示不返回任何结果。

- intp和uintp：指针大小的整数，分别表示签名和无签名类型。
- intc和uintc：相当于C语言的整型和无符号整型。
- int8、int16、int32和int64：固定宽度整型（无符号整型前面加u，比如uint8）。
- float32和float64：单精度和双精度浮点数类型。
- complex64和complex128：单精度和双精度复数类型。
- 数组可以用任何带索引的数值类型表示，比如float32[:]就是一维浮点数数组类型，int32[:,:]就是二维整型数组。

(2) 其他配置

除了及时编译，还有两个编译选项可以添加到@jit装饰器上。这两个选项将帮助我们完成Numba优化。选项具体描述如下。

(a) 没有GIL

无论何时，只要我们的代码用原始类型优化（不是用Python类型），GIL（第6章介绍过）就不再必要了。

有一种方法可以禁止GIL。我们可以把nogil=True属性传到装饰器。这样我们就可以用多线程运行Python代码（或者说是Numba代码）了。

也就是说，只要不再受GIL的限制，你就可以处理多线程系统的常见问题了（一致性、数据同步、竞态条件等）。

(b) 无Python模式

这个选项可以让我们设置Numba的编译模式。默认设置时，它将在模式之间跳转。Numba将针对需要优化的代码自动设置对应的优化模式。

一共有两种模式。一种是object模式。它产生的代码可以处理所有Python对象，并用C API完成Python对象上的操作。另一种是nopython模式，它可以不调用C API而生成更高效的代码。唯一的问题是，只有一部分函数和方法可以使用。

如果Numba不利用循环JIT（loop-jitting）方法，object模式就不会产生更快的代码（就是说循环可以被提取然后编译成nopython模式）。

我们可以用Numba把代码强制转换成nopython模式，如果转换失败就会产生错误。可以用下面的代码实现：

```
@jit(nopython=True)
def add2(a, b):
    return a + b
```

nopython模式的问题在于它有一些限制，除了这种模式支持的Python子集范围有限之外，还有：

- 函数里表示数值的所有原生类型都可以被引用
- 函数里不可以分配新内存

另外，由于使用循环JIT方式，被优化的循环内部不能产生返回状态。否则，这种情况不适合优化。

下面，让我们用一段示例代码来演示优化的过程：

```
def sum(x, y):
    array = np.arange(x * y).reshape(x, y)
    sum = 0
    for i in range(x):
        for j in range(y):
            sum += array[i, j]
    return sum
```

上面的代码取自Numba网站。这个函数适合循环JIT，也叫循环切换（loop-lifting）。为了让代码如预期运行，我们用Python REPL模式：

```
Python 2.7.9 |Continuum Analytics, Inc.| (default, Apr 14 2015, 12:54:25)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> from numba import jit
>>> import numpy as np
>>> @jit
... def sum_auto_jitting(x, y):
...     array = np.arange(x * y).reshape(x, y)
...     sum = 0
...     for i in range(x):
...         for j in range(y):
...             sum += array[i, j]
...     return sum
...
>>>
>>> sum_auto_jitting(2,65)
8385
>>> sum_auto_jitting.inspect_types()
```

另外，我们还直接使用了函数的inspect_types方法。这样做的好处是可以看到函数的源代码。这样在匹配Numba生成的机器码时，可以和源代码进行对照。

上面这个方法的输出结果可以帮助我们理解Numba优化背后的含义。更具体地说，可以看到具体的引用类型，是否进行了自动优化，以及每行Python代码被翻译成多少行代码。

让我们看看inspect_types方法可以从代码中获得哪些输出结果（这个结果会比REPL更具体）。



注意下面的结果是有删节版本。如果你想研究完整版，可以在自己的电脑上运行。

```
sum_auto_jitting (int64, int64)
--------------------------------------------------------------------------------
# File: <ipython-input-2-ff9f4104cbe2>
# --- LINE 1 ---

@jit

# --- LINE 2 ---

def sum_auto_jitting(x, y):

    # --- LINE 3 ---
    # label 0
    #   x = arg(0, name=x)  :: pyobject
    #   y = arg(1, name=y)  :: pyobject
    #   $0.1 = global(np: <module 'numpy' from
'/home/fernando/miniconda/lib/python2.7/site-packages/numpy/__init__.pyc'>)  ::
pyobject
    #   $0.2 = getattr(value=$0.1, attr=arange)  :: pyobject
    #   del $0.1
    #   $0.5 = x * y  :: pyobject
    #   $0.6 = call $0.2($0.5)  :: pyobject
    #   del $0.5
    #   del $0.2
    #   $0.7 = getattr(value=$0.6, attr=reshape)  :: pyobject
    #   del $0.6
    #   $0.10 = call $0.7(x, y)  :: pyobject
    #   del $0.7
    #   array = $0.10  :: pyobject
    #   del $0.10

    array = np.arange(x * y).reshape(x, y)

    # --- LINE 4 ---
    #   $const0.11 = const(int, 0)  :: pyobject
    #   sum = $const0.11  :: pyobject
    #   del $const0.11

    sum = 0

    # --- LINE 5 ---
    #   jump 40.1
    # label 40.1
    #   $const40.1.1 = const(LiftedLoop, LiftedLoop(<function sum_auto_jitting at
0x0000000007BB7F28>))  :: XXX Lifted Loop XXX
    #   $40.1.6 = call $const40.1.1(y, x, sum, array)  :: XXX Lifted Loop XXX
    #   del y
    #   del x
    #   del sum
```

```
    #   del array
    #   del $const40.1.1
    #   $40.1.8 = exhaust_iter(count=1, value=$40.1.6)  :: pyobject
    #   del $40.1.6
    #   $40.1.7 = static_getitem(index=0, value=$40.1.8)  :: pyobject
    #   del $40.1.8
    #   sum.1 = $40.1.7  :: pyobject
    #   del $40.1.7
    #   jump 103

    for i in range(x):

        # --- LINE 6 ---

        for j in range(y):

            # --- LINE 7 ---

            sum += array[i, j]

    # --- LINE 8 ---
    # label 103
    #   $103.2 = cast(value=sum.1)  :: pyobject
    #   del sum.1
    #   return $103.2

    return sum

# The function contains lifted loops
# Loop at line 5
# Has 1 overloads
# File: <ipython-input-2-ff9f4104cbe2>
# --- LINE 1 ---

@jit

# --- LINE 2 ---

def sum_auto_jitting(x, y):

    # --- LINE 3 ---

    array = np.arange(x * y).reshape(x, y)

    # --- LINE 4 ---

    sum = 0

    # --- LINE 5 ---
    # label 37
    #   y = arg(0, name=y)  :: int64
    #   x = arg(1, name=x)  :: int64
    #   sum = arg(2, name=sum)  :: int64
    #   array = arg(3, name=array)  :: array(int32, 2d, C)
```

```
    #   $37.1 = global(range: <class 'range'>)  :: range
    #   $37.3 = call $37.1(x)  :: (int64,) -> range_state_int64
    #   del x
    #   del $37.1
    #   $37.4 = getiter(value=$37.3)  :: range_iter_int64
    #   del $37.3
    #   $phi50.1 = $37.4  :: range_iter_int64
    #   del $37.4
    #   jump 50
    # label 50
    #   $50.2 = iternext(value=$phi50.1)  :: pair<int64, bool>
    #   $50.3 = pair_first(value=$50.2)  :: int64
    #   $50.4 = pair_second(value=$50.2)  :: bool
    #   del $50.2
    #   $phi53.1 = $50.3  :: int64
    #   del $50.3
    #   branch $50.4, 53, 102
    # label 53
    #   i = $phi53.1  :: int64
    #   del $phi53.1

    for i in range(x):

        # --- LINE 6 ---
        #   jump 56
        # label 56
        #   $56.1 = global(range: <class 'range'>)  :: range
        #   $56.3 = call $56.1(y)  :: (int64,) -> range_state_int64
        #   del $56.1
        #   $56.4 = getiter(value=$56.3)  :: range_iter_int64
        #   del $56.3
        #   $phi69.1 = $56.4  :: range_iter_int64
        #   del $56.4
        #   jump 69
        # label 69
        #   $69.2 = iternext(value=$phi69.1)  :: pair<int64, bool>
        #   $69.3 = pair_first(value=$69.2)  :: int64
        #   $69.4 = pair_second(value=$69.2)  :: bool
        #   del $69.2
        #   $phi72.1 = $69.3  :: int64
        #   del $69.3
        #   branch $69.4, 72, 98
        # label 72
        #   j = $phi72.1  :: int64
        #   del $phi72.1

        for j in range(y):

            # --- LINE 7 ---
            #  $72.6 = build_tuple(items=[Var(i, <ipython-input-2-ff9f4104cbe2> (5)),
Var(j, <ipython-input-2-ff9f4104cbe2> (6))])  :: (int64 x 2)
            #   del j
            #   $72.7 = getitem(index=$72.6, value=array)  :: int32
            #   del $72.6
```

7

```
            #   $72.8 = inplace_binop(immutable_fn=+, rhs=$72.7, lhs=sum, fn=+=)  ::
int64
            #    del $72.7
            #    sum = $72.8  :: int64
            #    del $72.8
            #    jump 69
            # label 98
            #    del i
            #    del $phi72.1
            #    del $phi69.1
            #    del $69.4
            #    jump 99
            # label 99
            #    jump 50
            # label 102
            #    del y
            #    del array
            #    del $phi53.1
            #    del $phi50.1
            #    del $50.4
            #    jump 103
            # label 103
            #   $103.2 = build_tuple(items=[Var(sum, <ipython-input-2-ff9f4104cbe2>
(5))])  :: (int64 x 1)
            #    del sum
            #    $103.3 = cast(value=$103.2)  :: (int64 x 1)
            #    del $103.2
            #    return $103.3

            sum += array[i, j]

    # --- LINE 8 ---

    return sum


================================================================================
```

要理解前面的输出结果的含义，需要注意看每一行源代码的编译过程都是由单独的行号开始的。后面跟着的是这一行生成的汇编指令，最后你可以看到不加注释的Python源代码。

注意看`LiftedLoop`行。在这一行你会看到Numba的优化代码。还需要注意在许多行最后引用的Numba类型。无论何时你看到一个`pyobject`类型，都不是原生类型，而是普通Python类型的封装版。

2. 在GPU上运行代码

前面已经提过，Numba代码可以运行在CPU和GPU上。其实，在GPU上运行并行程序可以进一步提升性能，比CPU上运行更快。

更具体地说就是，Numba支持CUDA编程（http://www.nvidia.com/object/cuda_home_new.html），按照CUDA模式的规则把一部分Python代码翻译成CUDA核心与设备支持的语言形式。

CUDA是Nvidia开发的并行计算平台和编程模式。它可以利用GPU获得更大的速度提升。

因为GPU编程是个较大的主题，可以写一整本书，所以这里不再介绍更多细节。我们只说明Numba具有这种能力，通过装饰器`@cuda.jit`就可以实现。关于这个主题的具体文档，请参考http://numba.pydata.org/numba-doc/0.18.2/cuda/index.html的内容。

7.2 pandas 工具

本章将介绍的第二个工具是pandas（http://pandas.pydata.org/）。它是一个开源的Python库，为用户提供高效、易用的数据结构和数据分析工具。

这个工具的诞生背景是，2008年程序员Wes McKinney在做财务数据量化分析的时候，需要一个高效的解决方法，于是发明了pandas。这个库已经成为Python社区中最流程和最活跃的项目之一。

用pandas写代码时在性能方面的一个亮点，是pandas的关键代码都是用Cython写的（第6章已经介绍过）。

7.2.1 安装 pandas

由于pandas很流行，所以有很多方法可以安装到你的系统上。具体方法由安装类型决定。

推荐直接用Anaconda的Python发行版进行安装（docs.continuum.io/anaconda/），里面包含pandas包和SciPy相关的包（比如NumPy、Matplotlib等）。这样，在你下载完之后，里面就已经有100多个包了，而且安装过程中也下载了100多兆字节的数据。

如果你不想安装Anaconda的完整版，可以使用`miniconda`包管理器（在前面介绍Numba的安装过程时已经介绍过）。采用这个方法时，你可以用`conda`命令进行安装。

(1) 用下面的代码创建一个新的Python虚拟环境：

```
$ conda create -n my_new_environment python
```

(2) 启动虚拟环境：

```
$ source activate my_new_environmen
```

(3) 最后安装pandas：

```
$ conda install pandas
```

另外，pandas也可以用pip命令行工具按照下面的命令进行安装（这可能也是最简单、兼容性最好的方法）：

```
$ pip install pandas
```

最后，还可以通过操作系统的包管理器进行安装。

Linux发行版	软件库链接	安装方法
Debian	packages.debian.org/search?keywords=pandas&searchon=names &suite=all&section=all	`$ sudo apt-get install python-pandas`
Ubuntu	http://packages.ubuntu.com/search?keywords=pandas &searchon=names&suite=all&section=all	`$ sudo apt-get install python-pandas`
OpenSUSE和Fedora	http://software.opensuse.org/package/python-pandas? search_term=pandas	`$ zypper in python-pandas`

如果你用前面几个安装方法都没安装成功，可以从官方网站http://pandas.pydata.org/pandas-docs/stable/install.html里找到安装方法。

7.2.2　用 pandas 做数据分析

在大数据和数据分析的世界里，知道正确的工具就是先手棋（当然，这只对了一半，还有一半是要学会使用工具）。对于数据分析和一些临时性的数据整理任务，常用手段是使用编程语言。编程语言比标准工具更具灵活性。

在数据分析领域，有两种语言主导这场性能竞赛：R和Python。Python可能让人震惊，因为前面我们已经看到，在数据处理方面Python的速度显然是不够快的。这就是pandas被创造出来的原因。

它提供了缓和并简化“数据争论”（data wrangling）任务的工具。

- 把大数据文件载入内存或保存为其他形式。
- 与matplotlib（http://matplotlib.org/）轻松整合，这样用几行代码就可以实现交互式的图形。
- 简单的语法可以实现数据补全、残缺字段等功能。

下面让我们用一个简单的例子来演示pandas如何在实现代码高性能的同时，还改善了代码的可读性。读取的文件是美国纽约市公共数据（NYC OpenData）网站（https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9）从2010年至今的311服务请求数据CSV文件（500M）。

代码试图通过纯Python和pandas两种形式统计文件中每个邮政编码的访问次数：

```
import pandas as pd
import time
```

```
import csv
import collections

SOURCE_FILE = './311.csv'


def readCSV(fname):
    with open(fname, 'rb') as csvfile:
        reader = csv.DictReader(csvfile)
        lines = [line for line in reader]
        return lines


def process(fname):
    content = readCSV(fname)
    incidents_by_zipcode = collections.defaultdict(int)
    for record in content:
        incidents_by_zipcode[toFloat(record['Incident Zip'])] += 1
    return sorted(incidents_by_zipcode.items(), reverse=True, key=lambda a:
int(a[1]))[:10]


def toFloat(number):
    try:
        return int(float(number))
    except:
        return 0


def process_pandas(fname):
    df = pd.read_csv(fname, dtype={
                     'Incident Zip': str, 'Landmark': str, 'Vehicle Type': str, 'Ferry
Direction': str})
    df['Incident Zip'] = df['Incident Zip'].apply(toFloat)
    column_names = list(df.columns.values)
    column_names.remove("Incident Zip")
    column_names.remove("Unique Key")
    return df.drop(column_names, axis=1).groupby(['Incident Zip'],
sort=False).count().sort('Unique Key', ascending=False).head(10)

init = time.clock()
total = process(SOURCE_FILE)
endtime = time.clock() - init
for item in total:
    print "%s\t%s" % (item[0], item[1])

print "(Pure Python) time: %s" % (endtime)

init = time.clock()
total = process_pandas(SOURCE_FILE)
endtime = time.clock() - init
print total
print "(Pandas) time: %s" % (endtime)
```

上面的`process`函数很简单，只有5行代码。它就是加载文件，做一些处理（主要是手工统计和汇总），对结果排序并提取前10个邮政编码。我们用的`defaultdict`数据类型是亮点，在前面的章节中为了优化函数性能曾经使用过。

另一方面，`process_pandas`函数做的事情基本相同，只不过是用pandas实现。有几行代码，不过都非常容易理解。它们显然都是“面向数据争论”（data-wrangling oriented）的，你会发现没有使用循环语句。我们可以通过字段名称自动接入每一列数据，然后对每组数据应用函数，中间不需要使用任何循环迭代。

前面的程序的运行结果如下图所示。

```
0       126452
11226   13525
10467   12044
10458   10413
10468   10028
10453   9622
11207   9391
11221   8974
10457   8896
11208   8883
(Pure Python) time: 13.068255
             Unique Key
Incident Zip
0                126452
11226             13525
10467             12044
10458             10413
10468             10028
10453              9622
11207              9391
11221              8974
10457              8896
11208              8883
(Pandas) time: 10.234595
```

你会发现用pandas简单地重新实现程序后，代码的运行时间减少了3秒钟。让我们继续深入研究pandas的API，以便进一步优化性能。代码还可以做两方面改进，而且都与使用了大量参数的`read_csv`方法有关。我们关心的两个参数如下。

- `usecols`：这个参数可以设置我们需要返回的列，帮助我们快速地从40多列中提取需要的两列数据。这样我们在返回结果之前就不需要再写删除多余列的逻辑了。
- `converters`：这个参数可以自动利用函数转换数据类型，不需要再使用`apply`方法进行转换。

我们的新函数如下所示：

```
def process_pandas(fname):
    df = pd.read_csv(fname, usecols=['Incident Zip', 'Unique Key'], converters={
                     'Incident Zip': toFloat}, dtype={'Incident Zip': str})
    return df.groupby(['Incident Zip'], sort=False).count().sort('Unique Key',
ascending=False).head(10)
```

就是这样，现在只需要两行代码！第一行的读取函数为我们做了大量工作，然后我们就是简单地分组、计数和排序。现在，让我们与前面的结果比较一下：

```
0        126452
11226    13525
10467    12044
10458    10413
10468    10028
10453    9622
11207    9391
11221    8974
10457    8896
11208    8883
(Pure Python) time: 13.059886
              Unique Key
Incident Zip
0                 126452
11226              13525
10467              12044
10458              10413
10468              10028
10453               9622
11207               9391
11221               8974
10457               8896
11208               8883
(Pandas) time: 3.030306
```

快10秒

我们的算法在性能上了10秒提升，需要处理的代码也大大减少，可以说是“双赢”。

代码还有一个优点，就是扩展性。pandas的函数可以在30秒内直接读取5.9G的数据文件。而我们的纯Python代码是不可能达到这一点的，如果没有足够的计算资源也不可能进行数据处理。

7.3 Parakeet

这是一个最明确的Python数据处理工具。说它明确是因为它只支持一小部分Python和NumPy组合的数据类型。因此，如果你想解决更普遍的任务，它可能不是你的选择，但是如果你觉得可以用它解决问题，就继续往下看。

要更具体地了解Parakeet的限制（通常只在数值计算方面有效），我们总结了一个列表。

- 支持的数据类型包括Python的数字、元组、列表和NumPy的数组。
- Parakeet会自动对数据类型执行向上转换，就是说，无论何时遇到两种不同类型的数据，都会被强制向上转换成统一类型。例如，Python表达式`1.0 if b else false`会被转换成`1.0 if b else 0.0`，但是当自动转换失败时，例如`1.0 if b else (1,2)`，在编译过程中就会出现转换错误（见下一条）。
- Parakeet里面不能捕捉和处理异常；也不支持`break`和`continue`语句。这是因为Parakeet是用SSA结构展示程序的（http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.4503）。

- 数组传播（array broadcasting，NumPy的特性）是通过对数组参数类型显式地映射操作实现的。这种实现方式的限制是它不能实现多维数组传播（比如8×2×3和7×2数组）。
- 只实现了一小部分Python和NumPy的内置函数。完整的函数列表请见https://github.com/iskandr/parakeet/blob/master/parakeet/mappings.py。
- 列表综合表达式作为数组综合表达式处理。

7.3.1　安装 Parakeet

Parakeet的安装十分简单。如果你用pip安装也不难。简单地输入下面命令即可：

```
$ pip install parakeet
```

这样就搞定了！

如果你想通过源代码安装，可能需要提前安装一些依赖。安装包列表如下所示：

- Python 2.7
- dsltools（https://github.com/iskandr/dsltools）
- nose，用于测试（https://nose.readthedocs.org/en/latest/）
- NumPy（http://www.scipy.org/install.html）
- appDirs（https://pypi.python.org/pypi/appdirs/）
- gcc 4.4+，用于OpenMP后端支持的默认值



如果你运行的是Windows系统，32位系统版本有官方支持。如果是64位系统就没那么好运了，没有官方支持的文档。

如果你是OS X用户，可能需要用HomeBrew安装一个新版的C编译器，因为clang和gcc都可能没有及时更新。

当安装完依赖之后，从https://github.com/iskandr/parakeet下载源代码，并运行下面的命令即可（在代码文件夹内）：

```
$ python setup.py install
```

7.3.2　Parakeet 是如何工作的

下面不再深入介绍Parakeet背后的理论细节，让我们看看如何用它来优化代码。这样，不翻文档，你也能了解这个模块。

这个库的主要结构是在函数上使用的一个装饰器，因此Parakeet会尽可能地控制和优化你的

代码。

为了完成简单的测试，我们用Parakeet网站上的一个简单示例来演示，计算一个4000*4000随机浮点数列表。代码将用两种方式执行同样的函数，一种是Parakeet方式，另一种是未优化的方式。然后会测量两种方式处理同样输入数据的运行时间：

```
from parakeet import jit
import random
import numpy as np
import time


@jit
def allpairs_dist_prkt(X, Y):
    def dist(x, y):
        return np.sum((x-y)**2)
    return np.array([[dist(x, y) for y in Y] for x in X])


def allpairs_dist_py(X, Y):
    def dist(x, y):
        return np.sum((x-y)**2)
    return np.array([[dist(x, y) for y in Y] for x in X])

input_a = [random.random() for x in range(0, 10000)]
input_b = [random.random() for x in range(0, 10000)]

print "----------------------------------------------"
init = time.clock()
allpairs_dist_py(input_a, input_b)
end = time.clock()
print "Total time pure python: %s" % (end - init)

print
init = time.clock()
allpairs_dist_prkt(input_a, input_b)
end = time.clock()
print "Total time parakeet: %s" % (end - init)
print "----------------------------------------------"
```

在一个i7处理器、8G RAM的电脑上，我们可以获得的运行时间如下：

```
---------------------------------------------
Total time pure python: 73.19119

Total time parakeet: 0.088978
---------------------------------------------
```

上图显示了通过这个函数性能得到的显著提升（通过Parakeet编译了一部分Python类型）。

装饰器函数是为一些函数提供的模板，一个类型一个（在我们的例子中，只有一个）。新函数在被Parakeet翻译成机器码之前，可以通过不同的方式进行优化。



需要注意的是，即性能得到了优化，Parakeet也只支持一部分Python数据类型，所以它不能作为通用的性能优化手段（实际情况是Parakeet的使用场景十分有限）。

7.4　小结

在这一章里，我们介绍了Python的三种数据处理工具，并且介绍了具体的使用场景（但是性能卓越），比如Parakeet，以及更通用的Numba和pandas。对于这三个工具，我们都介绍了相关的基础知识：简介、安装和示例。它们还有很多功能，你可以根据自己的需求去发掘。不过，本章的内容足以让你找到正确的方向。

下一章也是最后一章，我们将介绍一个实际的性能优化示例。我们将把在本书中学到的所有知识都应用其中。

第8章 付诸实践

欢迎来到本书的最后一章。如果你一直看到这里，一定已经学会了不少优化技术，既有专门针对Python编程语言的技术，也有可用于其他语言的通用技术。

你也学习过性能分析和数据可视化工具。我们还专门介绍过Python在科学数据处理领域的优化技术。掌握了这些工具，你就可以优化代码的性能。

在这最后一章，我们将介绍一个实际的示例，其中会涵盖前几章介绍过的所有技术（不过前面介绍的部分工具可以相互替代，所以全部都用不见得会取得更好的效果）。我们将编写一段初始代码，测量其性能，然后通过优化过程重写代码，并重新测量性能。

8.1 需要解决的问题

在开始思考编写初始代码之前，让我们介绍一下需要解决的问题。

鉴于本书的范围有限，用一个大而全的示例可能不太合适，所以我们重点解决一个小问题。小问题可以让我们有的放矢，而且也可以避免在这么短的时间内承担大量的优化任务。

为了增加趣味性，我们把问题分成了两部分。

- **第一部分**：这部分的主要任务是找出要处理的数据。我们并非简单地从URL下载数据，而是从网站上抓取。
- **第二部分**：这部分的重点是处理在第一部分获得的数据。在这一步，我们将实现大量的CPU相关的操作，统计我们收集的数据。

对于这两部分内容，我们首先都会给出解决问题的原始代码，暂时不考虑性能问题。之后，我们会分别对两个解决方案进行分析，并尽可能地改善性能。

8.1.1 从网站上抓取数据

我们要抓取的网站是科幻与灵异网（Science Fiction & Fantasy，http://scifi.stackexchange.com/）。这个网站主要是回答与科幻和灵异有关的问题。它类似于StackOverflow，不过主要是面向科幻与灵异爱好者。

更具体地说，我们想抓取最新的问题列表。对于每个问题，我们获取带有问题和所有答案的页面。当所有的抓取和分析工作完成之后，我们会把相关的信息保存为JSON格式，方便后面进行分析。

虽然我们要处理HTML页面，但是我们并不需要它们。所以我们要去掉所有的HTML代码，只保存下面的内容：

- 问题的标题
- 问题的作者
- 问题的内容（问题的具体文字）
- 答案的内容（如果存在）
- 答案的作者

针对这些信息，我们可以做一些有趣的后期处理，并获取一些相关的统计结果（稍后详细讨论）。

程序最终的输出结果应该会像下面这样：

```
{
    "questions": [{
        "title": "Ending of John Carpenter's The Thing",
        "body": "In the ending of John Carpenter's classic 1982 sci-
        fi horror film The Thing,
        is...
        ",
        "author": "JMFB",
        "answers": [{
            "body": "This is the million dollar question, ...
            Unfortunately,
            he is notoriously...
            ",
            "author": "Richard",
        }, {
            "body": "Not to point out what may seem obvious,
            but Childs isn 't breathing. Note the total absence of ",
            "author": "user42"
        }]
    }, {
        "title": "Was it ever revealed what pedaling the bicycles in
        the second episode was doing ? ",
        "body" : "I'm going to assume they were probably some sort of
        turbine...electricity...something,
        but I 'd prefer to know
        for sure.
        ",
        "author": "bartz",
        "answers": [{
            "body": "The Wikipedia synopsis states: most citizens
            make a living pedaling exercise bikes all day in order
            to generate power
            for their environment ",
            "author": "Jack Nimble"
```

```
        }]
    }]
}
```

这个脚本会把所有的信息都保存到一个JSON文件中，具体字段会在代码中预先定义好。

我们要让初始的两个脚本都尽量简单。也就是说，要使用最少的模块。在这个案例中，我们使用的主要模块如下所示。

- Beautiful Soup（http://www.crummy.com/software/BeautifulSoup/）：这个模块用于解析HTML文件，主要是因为它提供了一整套解析网页的API，可以自动识别页面编码格式（如果你处理页面编码格式的经验很丰富，可能会嫌弃这个功能），以及使用CSS样式选择器（selector）遍历HTML结构树。
- Requests（http://docs.python-requests.org/en/latest/）：这个模块用于生成HTTP请求。虽然Python已经提供了相应的模块，但是这个模块简化了API，而且通过更加具有Python风格的方式实现网络请求。

你可以通过pip命令行工具安装两个模块：

```
$ pip install requests beautifulsoup4
```

我们为了获取数据将要抓取和解析的页面如下图所示。



8.1.2　数据预处理

第二个脚本是读取JSON文件，然后进行一些统计分析。为了获得更有趣的分析结果，我们不只是统计每个用户提问的次数（虽然我们也会获得这个数据）。我们还要计算下面的信息：

- 提问最多的10名用户
- 回答最多的10名用户
- 最常问的主题
- 最短的答案
- 最常用的10个短语
- 答案最多的10个问题

由于本书的主题是性能，而不是自然语言处理（Natural Language Processing，NLP），所以我们不会深入介绍脚本中关于NLP的部分代码。我们将集中精力利用目前所学的Python优化方法改善代码的性能。

我们在这个脚本中使用的唯一一个非标准模块是NLTK（http://www.nltk.org/），用来实现自然语言处理的功能。

8.2　编写初始代码

根据前面的描述，让我们把所有将来要优化的代码都列出来。

下面几点的第一条非常简单：一个单文件脚本，可以像前面介绍的那样，抓取数据并保存为JSON格式。流程很简单，顺序如下所示。

(1) 逐页查询问题列表。
(2) 收集每一页的问题链接。
(3) 收集每一个链接指向的页面里的信息。
(4) 移动到下一页，重复前面3点。
(5) 最终把所有数据保存为JSON格式。

代码如下所示：

```
from bs4 import BeautifulSoup
import requests
import json


SO_URL = "http://scifi.stackexchange.com"
QUESTION_LIST_URL = SO_URL + "/questions"
MAX_PAGE_COUNT = 2
```

```
global_results = []
initial_page = 1  # 首页就是第一页


def get_author_name(body):
    link_name = body.select(".user-details a")
    if len(link_name) == 0:
        text_name = body.select(".user-details")
        return text_name[0].text if len(text_name) > 0 else 'N/A'
    else:
        return link_name[0].text


def get_question_answers(body):
    answers = body.select(".answer")
    a_data = []
    if len(answers) == 0:
        return a_data

    for a in answers:
        data = {
            'body': a.select(".post-text")[0].get_text(),
            'author': get_author_name(a)
        }
        a_data.append(data)
    return a_data


def get_question_data(url):
    print "Getting data from question page: %s " % (url)
    resp = requests.get(url)
    if resp.status_code != 200:
        print "Error while trying to scrape url: %s" % (url)
        return
    body_soup = BeautifulSoup(resp.text)
    # 定义一个将被转换成JSON格式的输出词典
    q_data = {
        'title': body_soup.select('#question-header .question-hyperlink')[0].text,
        'body': body_soup.select('#question .post-text')[0].get_text(),
        'author': get_author_name(body_soup.select(".post-signature.owner")[0]),
        'answers': get_question_answers(body_soup)
    }
    return q_data


def get_questions_page(page_num, partial_results):
    print "======================================================"
    print " Getting list of questions for page %s" % (page_num)
    print "======================================================"

    url = QUESTION_LIST_URL + "?sort=newest&page=" + str(page_num)
    resp = requests.get(url)
    if resp.status_code != 200:
```

```
        print "Error while trying to scrape url: %s" % (url)
        return
    body = resp.text
    main_soup = BeautifulSoup(body)

    # 获取每个问题的网络链接
    questions = main_soup.select('.question-summary .question-hyperlink')
    urls = [SO_URL + x['href'] for x in questions]
    for url in urls:
        q_data = get_question_data(url)
        partial_results.append(q_data)
    if page_num < MAX_PAGE_COUNT:
        get_questions_page(page_num + 1, partial_results)


get_questions_page(initial_page, global_results)
with open('scrapping-results.json', 'w') as outfile:
    json.dump(global_results, outfile, indent=4)

print '----------------------------------------------------'
print 'Results saved'
```

检查前面的代码，你会发现我们实现了目标。现在，我们只使用了几个必要的外部库，以及Python自带的JSON模块。

另一方面，第二个脚本被分割成两部分，以方便组织。

- analyzer.py：这个文件里包含关键代码。其作用是把JSON文件加载到一个`dict`结构中，执行一系列计算。
- visualizer.py：这个文件里简单地包含了一组函数，用来可视化分析结果。

让我们看看这两个文件的代码。第一组函数的功能是用来完成清洗数据、加载到内存等操作：

```
#analyzer.py
import operator
import string
import nltk
from nltk.util import ngrams
import json
import re
import visualizer

SOURCE_FILE = './scrapping-results.json'

#加载JSON文件并定义输出词典
def load_json_data(file):
    with open(file) as input_file:
        return json.load(input_file)

def analyze_data(d):
    return {
```

```
        'shortest_answer': get_shortest_answer(d),
        'most_active_users': get_most_active_users(d, 10),
        'most_active_topics': get_most_active_topics(d, 10),
        'most_helpful_user': get_most_helpful_user(d, 10),
        'most_answered_questions': get_most_answered_questions(d, 10),
        'most_common_phrases':  get_most_common_phrases(d, 10, 4),
    }

# 把问题的正文内容组合成一个列表
def flatten_questions_body(data):
    body = []
    for q in data:
        body.append(q['body'])
    return '. '.join(body)

# 把问题的标题内容组合成一个小写的列表
def flatten_questions_titles(data):
    body = []
    pattern = re.compile('(\[|\])')
    for q in data:
        lowered = string.lower(q['title'])
        filtered = re.sub(pattern, ' ', lowered)
        body.append(filtered)
    return '. '.join(body)
```

下面一组函数用来完成数据计数（counting），然后通过不同的方式分析JSON数据获取统计结果：

```
# 返回提问问题数量最多的作者排名
def get_most_active_users(data, limit):
    names = {}
    for q in data:
        if q['author'] not in names:
            names[q['author']] = 1
        else:
            names[q['author']] += 1
    return sorted(names.items(), reverse=True, key=operator.itemgetter(1))[:limit]


def get_node_content(node):
    return ' '.join([x[0] for x in node])

# 返回问题标题中最常见的主题排名
def get_most_active_topics(data, limit):
    body = flatten_questions_titles(data)
    sentences = nltk.sent_tokenize(body)
    sentences = [nltk.word_tokenize(sent) for sent in sentences]
    sentences = [nltk.pos_tag(sent) for sent in sentences]
    grammar = "NP: {<JJ>?<NN.*>}"
    cp = nltk.RegexpParser(grammar)
    results = {}
    for sent in sentences:
        parsed = cp.parse(sent)
```

```
        trees = parsed.subtrees(filter=lambda x: x.label() == 'NP')
        for t in trees:
            key = get_node_content(t)
            if key in results:
                results[key] += 1
            else:
                results[key] = 1
    return sorted(results.items(), reverse=True,
key=operator.itemgetter(1))[:limit]

# 返回答题最多的用户排名
def get_most_helpful_user(data, limit):
    helpful_users = {}
    for q in data:
        for a in q['answers']:
            if a['author'] not in helpful_users:
                helpful_users[a['author']] = 1
            else:
                helpful_users[a['author']] += 1

    return sorted(helpful_users.items(), reverse=True,
key=operator.itemgetter(1))[:limit]

# 返回答案最多的问题排名
def get_most_answered_questions(d, limit):
    questions = {}

    for q in d:
        questions[q['title']] = len(q['answers'])
    return sorted(questions.items(), reverse=True,
key=operator.itemgetter(1))[:limit]

# 返回使用最常用词组数量最多的问题排名
def get_most_common_phrases(d, limit, length):
    body = flatten_questions_body(d)
    phrases = {}
    for sentence in nltk.sent_tokenize(body):
        words = nltk.word_tokenize(sentence)
        for phrase in ngrams(words, length):
            if all(word not in string.punctuation for word in phrase):
                key = ' '.join(phrase)
                if key in phrases:
                    phrases[key] += 1
                else:
                    phrases[key] = 1

    return sorted(phrases.items(), reverse=True,
key=operator.itemgetter(1))[:limit]

# 返回答案最短的问题排名
def get_shortest_answer(d):

    shortest_answer = {
        'body': '',
```

```
        'length': -1
    }
    for q in d:
        for a in q['answers']:
            if len(a['body']) < shortest_answer['length'] or
shortest_answer['length'] == -1:
                shortest_answer = {
                    'question': q['body'],
                    'body': a['body'],
                    'length': len(a['body'])
                }
    return shortest_answer
```

下面的代码显示如何使用前面声明的函数，并显示函数的结果。处理它们都需要遵循下面三个步骤。

(1) 把JSON文件载入内存。
(2) 处理数据，然后把结果保存到词典中。
(3) 遍历词典，显示结果。

前面的步骤用下面的代码实现：

```
data_dict = load_json_data(SOURCE_FILE)

results = analyze_data(data_dict)

print "=== ( Shortest Answer ) === "
visualizer.displayShortestAnswer(results['shortest_answer'])

print "=== ( Most Active Users ) === "
visualizer.displayMostActiveUsers(results['most_active_users'])

print "=== ( Most Active Topics ) === "
visualizer.displayMostActiveTopics(results['most_active_topics'])

print "=== ( Most Helpful Users ) === "
visualizer.displayMostHelpfulUser(results['most_helpful_user'])

print "=== ( Most Answered Questions ) === "
visualizer.displayMostAnsweredQuestions(results['most_answered_questions'])

print "=== ( Most Common Phrases ) === "
visualizer.displayMostCommonPhrases(results['most_common_phrases'])
```

下面文件中的代码主要是为了把输出结果表现成人性化的形式：

```
#visualizer.py
def displayShortestAnswer(data):
    print "A: %s" % (data['body'])
    print "Q: %s" % (data['question'])
    print "Length: %s characters" % (data['length'])
```

```
def displayMostActiveUsers(data):
    index = 1
    for u in data:
        print "%s - %s (%s)" % (index, u[0], u[1])
        index += 1

def displayMostActiveTopics(data):
    index = 1
    for u in data:
        print "%s - %s (%s)" % (index, u[0], u[1])
        index += 1

def displayMostHelpfulUser(data):
    index = 1
    for u in data:
        print "%s - %s (%s)" % (index, u[0], u[1])
        index += 1

def displayMostAnsweredQuestions(data):
    index = 1
    for u in data:
        print "%s - %s (%s)" % (index, u[0], u[1])
        index += 1

def displayMostCommonPhrases(data):
    index = 1
    for u in data:
        print "%s - %s (%s)" % (index, u[0], u[1])
        index += 1
```

8.2.1　分析代码性能

代码性能分析可以通过下面两步来完成，就像我们之前做过的那样。对每个文件，我们都分析代码性能，获取数据，然后寻找优化方法，最后改写代码并重新测量代码性能。



就像前面代码优化的过程中重复的几个步骤一样：性能分析—改写代码—再次性能分析，我们将用有限的步骤来获得结果。但是，切记优化过程十分漫长，没有尽头。

网络爬虫代码优化

在开启优化过程之前，让我们先获取一些测量数据，以便进行对比。

一个容易获得的数据是程序运行的总时间（在我们的例子中，为了简化，我们限制页面查询总数为20）。

简单运行`time`命令行工具，就可以获得程序的运行时间：

```
$ time python scraper.py
```

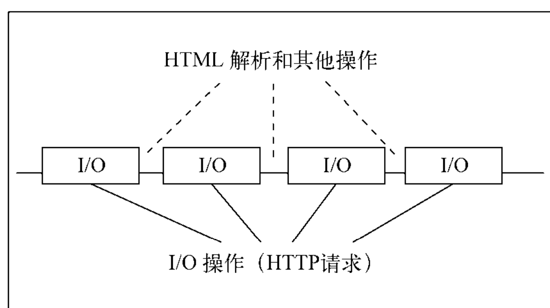
下面的截图表明，抓取并解析20个页面的问题数据，并转换成3MB的JSON文件，一共要用7分半钟。

```
--------------------
Results saved

real    7m37.963s
user    0m57.186s
sys     0m0.759s
```

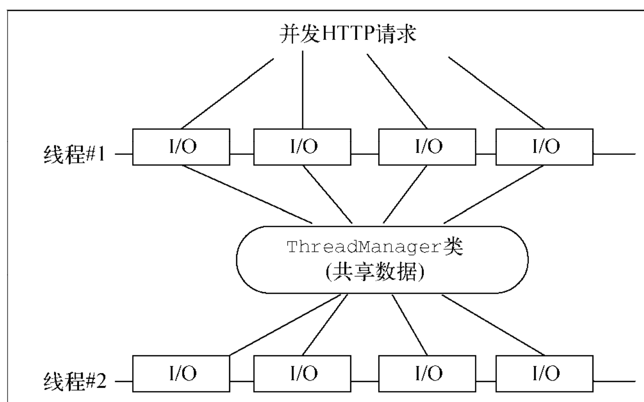
网络爬虫脚本基本算是一个IO密集型的循环任务，通过最少的处理步骤从互联网上获取数据。因此，我们可以找到的第一条也是最符合逻辑的一条优化需求，就是网络爬虫没有并行地处理请求。由于我们的代码不是CPU密集型，所以我们可以安全地使用多线程模块（请参考第5章），通过最少的努力获取最大的性能提升。

为了说清楚我们要做的事情，用下面的图形表示当前的爬虫脚本：



我们花了大量的时间运行I/O操作，更具体地说，我们通过HTTP请求获取每一个页面的问题列表。

就像我们之前看到的，I/O密集型操作通过多线程模块实现并行十分简单。因此，我们要把脚本转换成下面的形式。



现在，让我们看看优化过的代码。我们首先会看到ThreadManager类，它主要用于集中管理线程的配置，以及整个并行进程的状态：

```
from bs4 import BeautifulSoup
import requests
import json
import threading


SO_URL = "http://scifi.stackexchange.com"
QUESTION_LIST_URL = SO_URL + "/questions"
MAX_PAGE_COUNT = 20


class ThreadManager:
    instance = None
    final_results = []
    threads_done = 0
    # 并行线程的数量，
    # 将决定每个线程获取的页面总数量
    totalConnections = 4

    @staticmethod
    def notify_connection_end(partial_results):
        print "==== Thread is done! ====="
        ThreadManager.threads_done += 1
        ThreadManager.final_results += partial_results
        if ThreadManager.threads_done == ThreadManager.totalConnections:
            print "==== Saving data to file! ===="
            with open('scrapping-results-optimized.json', 'w') as outfile:
                json.dump(ThreadManager.final_results, outfile, indent=4)
```

下面的函数通过BeatifulSoup模块抓取页面信息，获取页面里的问题列表，或者获取每个问题的具体信息：

```
def get_author_name(body):
    link_name = body.select(".user-details a")
```

```
    if len(link_name) == 0:
        text_name = body.select(".user-details")
        return text_name[0].text if len(text_name) > 0 else 'N/A'
    else:
        return link_name[0].text


def get_question_answers(body):
    answers = body.select(".answer")
    a_data = []
    if len(answers) == 0:
        return a_data

    for a in answers:
        data = {
            'body': a.select(".post-text")[0].get_text(),
            'author': get_author_name(a)
        }
        a_data.append(data)
    return a_data


def get_question_data(url):
    print "Getting data from question page: %s " % (url)
    resp = requests.get(url)
    if resp.status_code != 200:
        print "Error while trying to scrape url: %s" % (url)
        return
    body_soup = BeautifulSoup(resp.text)
    # 定义一个将被转换成JSON格式的输出词典
    q_data = {
        'title': body_soup.select('#question-header .question-hyperlink')[0].text,
        'body': body_soup.select('#question .post-text')[0].get_text(),
        'author': get_author_name(body_soup.select(".post-signature.owner")[0]),
        'answers': get_question_answers(body_soup)
    }
    return q_data


def get_questions_page(page_num, end_page, partial_results):
    print "======================================================"
    print " Getting list of questions for page %s" % (page_num)
    print "======================================================"

    url = QUESTION_LIST_URL + "?sort=newest&page=" + str(page_num)
    resp = requests.get(url)
    if resp.status_code != 200:
        print "Error while trying to scrape url: %s" % (url)
    else:
        body = resp.text
        main_soup = BeautifulSoup(body)

        # 获取每个问题的网络链接
        questions = main_soup.select('.question-summary .question-hyperlink')
```

8

```
        urls = [SO_URL + x['href'] for x in questions]
        for url in urls:
            q_data = get_question_data(url)
            partial_results.append(q_data)
    if page_num + 1 < end_page:
        get_questions_page(page_num + 1,  end_page, partial_results)
    else:
        ThreadManager.notify_connection_end(partial_results)

pages_per_connection = MAX_PAGE_COUNT / ThreadManager.totalConnections
for i in range(ThreadManager.totalConnections):
    init_page = i * pages_per_connection
    end_page = init_page + pages_per_connection
    t = threading.Thread(target=get_questions_page,
                         args=(init_page, end_page, [],),
                         name='connection-%s' % (i))
    t.start()
```

从pages_per_connection开始到t.start()的这部分代码，是与前一个版本差别最大的地方。现在不是从第1页开始，然后一页一页地抓取了，而是从预先设置好的线程数量（直接使用threading.Thread类）开始，并行地调用我们的get_question_page函数。我们要做的就是把这个函数作为target传给线程。

之后，我们还需要一个方式来集中管理线程的配置参数和每个线程产生的临时结果。因此，我们创建了ThreadManager类。

如下图所示，代码改变之后，运行时间从原来的7分半钟降低到了2分13秒。

```
real    2m13.450s
user    1m9.068s
sys     0m5.031s
```

调整线程的数量，可能会获取更快的运行速度，但是主要的优化方法就是这样。

8.2.2　数据分析代码的优化

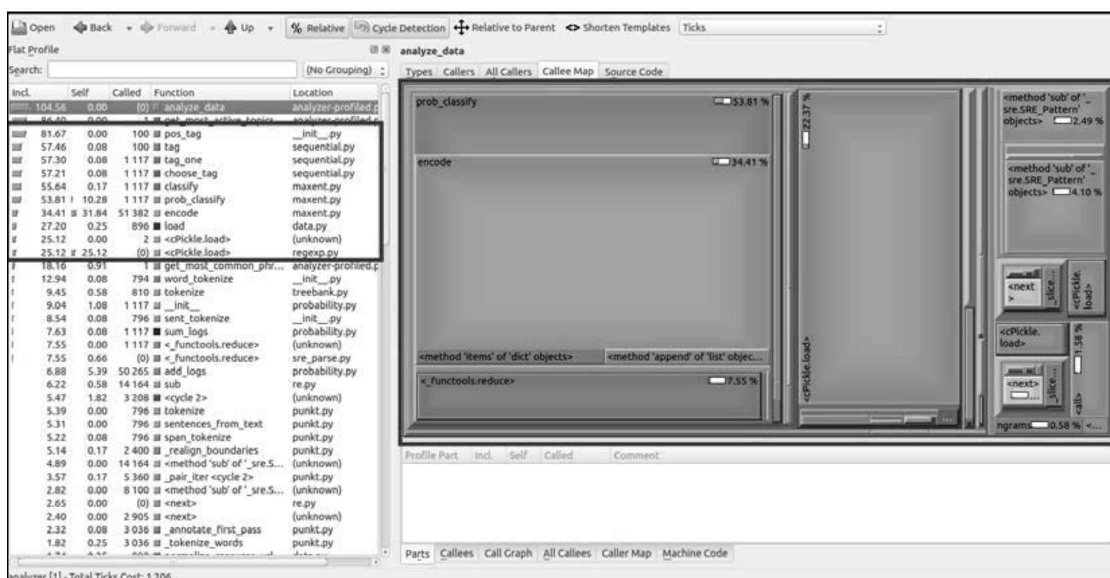
数据分析脚本与网络爬虫脚本不同。它不是一个I/O密集型的脚本，而是CPU密集型脚本。它需要的I/O操作极少，主要是读取文件，输出结果。因此，我们重点测量与CPU相关的细节。

让我们首先获取一些基本的测量数据，以便有个基本认识。

```
real    0m3.470s
user    0m1.324s
sys     0m0.084s
```

上图显示了time命令行工具测量的结果。现在我们有一个基本的数据做参照了，我们知道自己的目标是把运行时间降到3.5秒以下。

第一种方法是通过`cProfile`性能分析器获取代码运行的时间数据。这可以帮我们全面地认识整个程序，捕捉到性能分析的切入点。分析结果如下图所示。



从上面的截图中可以看出两点。

- ❑ 在左边的列表中，我们可以看出函数需要消耗的时间。我们会发现列表中最耗时的函数大都源自`nltk`模块（前两个函数的时间都是下面的函数消耗的，所以它们俩其实不重要）。
- ❑ 在右边图中，函数调用图（Callee Map）看起来非常复杂，难以解释（和实际情况差别很大，里面出现的大多数函数都不在代码中，而是在我们使用的模块里）。

也就是说，直接改善我们的代码并不是很简单。我们可能需要换个思路：由于我们做了大量的计算，可能使用带类型的代码会有用。因此，让我们用Cython试试。

使用Cython命令行工具分析我们的代码，会发现大部分代码都不能直接编译成C语言。分析结果如下图所示。

```
Generated by Cython 0.22

Raw output: analyzer.c

+001: import operator
+002: import string
+003: import nltk
+004: from nltk.util import ngrams
+005: import json
+006: import re
+007: import visualizer
 008:
 009:
+010: SOURCE_FILE = './scrapping-results.json'
 011:
 012: # Returns the top "limit" users with the most questions asked
+013: def get_most_active_users(data, limit):
+014:   names = {}
+015:   for q in data:
+016:           if q['author'] not in names:
+017:                   names[q['author']] = 1
 018:           else:
+019:                   names[q['author']] += 1
+020:   return sorted(names.items(), reverse=True, key=operator.itemgetter(1))[:limit]
 021:
+022: def get_node_content(node):
+023:   return ' '.join([x[0] for x in node])
 024:
 025: # Tries to extract the most common topics from the question's titles
+026: def get_most_active_topics(data, limit):
+027:   body = flatten_questions_titles(data)
+028:   sentences = nltk.sent_tokenize(body)
+029:   sentences = [nltk.word_tokenize(sent) for sent in sentences]
+030:   sentences = [nltk.pos_tag(sent) for sent in sentences]
+031:   grammar = "NP: {<JJ>?<NN.*>}"
+032:   cp = nltk.RegexpParser(grammar)
+033:   results = {}
+034:   for sent in sentences:
+035:           parsed = cp.parse(sent)
+036:           trees = parsed.subtrees(filter=lambda x: x.label() == 'NP')
+037:           for t in trees:
+038:                   key = get_node_content(t)
+039:                   if key in results:
+040:                           results[key] += 1
 041:                   else:
+042:                           results[key] = 1
+043:   return sorted(results.items(), reverse=True, key=operator.itemgetter(1))[:limit]
 044:
 045: # Returns the user that has the most answers
+046: def get_most_helpful_user(data, limit):
```

上面的截图显示了我们的代码中需要分析的部分。可以清晰地看出深色代码满屏都是，这说明大部分代码都没有直接翻译成C语言。可惜的是，由于我们在绝大多数函数中都处理了一个复杂的对象，所以我们能优化的地方不多。

另外，简单地用Cython编译我们的代码，其实就可以获得更好的结果。因此，让我们看看应该如何调整源代码，以便可以用Cython编译它。第一个文件基本上和原始的数据分析代码一样，只是改变了高亮显示的代码，减少了一些函数的调用，而且把代码转成了一个外部库：

```
#analyzer_cython.pyx
import operator
import string
import nltk
from nltk.util import ngrams
import json
import re


SOURCE_FILE = './scrapping-results.json'
```

```
# 返回提问问题数量最多的作者排名
def get_most_active_users(data, int limit):
    names = {}
    for q in data:
        if q['author'] not in names:
            names[q['author']] = 1
        else:
            names[q['author']] += 1
    return sorted(names.items(), reverse=True, key=operator.itemgetter(1))[:limit]


def get_node_content(node):
    return ' '.join([x[0] for x in node])

# 返回问题标题中最常见的主题排名
def get_most_active_topics(data, int limit):
    body = flatten_questions_titles(data)
    sentences = nltk.sent_tokenize(body)
    sentences = [nltk.word_tokenize(sent) for sent in sentences]
    sentences = [nltk.pos_tag(sent) for sent in sentences]
    grammar = "NP: {<JJ>?<NN.*>}"
    cp = nltk.RegexpParser(grammar)
    results = {}
    for sent in sentences:
        parsed = cp.parse(sent)
        trees = parsed.subtrees(filter=lambda x: x.label() == 'NP')
        for t in trees:
            key = get_node_content(t)
            if key in results:
                results[key] += 1
            else:
                results[key] = 1
    return sorted(results.items(), reverse=True,
key=operator.itemgetter(1))[:limit]


# 返回答题最多的用户排名
def get_most_helpful_user(data, int limit):
    helpful_users = {}
    for q in data:
        for a in q['answers']:
            if a['author'] not in helpful_users:
                helpful_users[a['author']] = 1
            else:
                helpful_users[a['author']] += 1

    return sorted(helpful_users.items(), reverse=True,
key=operator.itemgetter(1))[:limit]


# 返回答案最多的问题排名
def get_most_answered_questions(d, int limit):
    questions = {}
```

8

```
    for q in d:
        questions[q['title']] = len(q['answers'])
    return sorted(questions.items(), reverse=True,
key=operator.itemgetter(1))[:limit]


# 把问题的正文内容组合成一个列表
def flatten_questions_body(data):
    body = []
    for q in data:
        body.append(q['body'])
    return '. '.join(body)


# 把问题的标题内容组合成一个小写的列表
def flatten_questions_titles(data):
    body = []
    pattern = re.compile('(\[|\])')
    for q in data:
        lowered = string.lower(q['title'])
        filtered = re.sub(pattern, ' ', lowered)
        body.append(filtered)
    return '. '.join(body)


# 返回使用最常用词组数量最多的问题排名
def get_most_common_phrases(d, int limit, int length):
    body = flatten_questions_body(d)
    phrases = {}
    for sentence in nltk.sent_tokenize(body):
        words = nltk.word_tokenize(sentence)
        for phrase in ngrams(words, length):
            if all(word not in string.punctuation for word in phrase):
                key = ' '.join(phrase)
                if key in phrases:
                    phrases[key] += 1
                else:
                    phrases[key] = 1

    return sorted(phrases.items(), reverse=True,
key=operator.itemgetter(1))[:limit]


# 返回答案最短的问题排名
def get_shortest_answer(d):
    cdef int shortest_length = 0

    shortest_answer = {
        'body': '',
        'length': -1
    }
    for q in d:
        for a in q['answers']:
```

```
            if len(a['body']) < shortest_length or shortest_length == 0:
                shortest_length = len(a['body'])
                shortest_answer = {
                    'question': q['body'],
                    'body': a['body'],
                    'length': shortest_length
                }
    return shortest_answer


# 加载JSON文件并返回输出结果的词典
def load_json_data(file):
    with open(file) as input_file:
        return json.load(input_file)


def analyze_data(d):
    return {
        'shortest_answer': get_shortest_answer(d),
        'most_active_users': get_most_active_users(d, 10),
        'most_active_topics': get_most_active_topics(d, 10),
        'most_helpful_user': get_most_helpful_user(d, 10),
        'most_answered_questions': get_most_answered_questions(d, 10),
        'most_common_phrases':  get_most_common_phrases(d, 10, 4),
    }
```

下面的文件是Cython编译源代码，我们在前面已经看到过这段代码（请参考第6章）：

```
#analyzer-setup.py
from distutils.core import setup
from Cython.Build import cythonize

setup(
    name = 'Analyzer app',
    ext_modules = cythonize("analyzer_cython.pyx"),
)
```

最后一个文件把我们新改的外部库导入到编译的模块中。这个文件会调用load_json_data和analyze_data方法，最后使用visualizer模块生成输出结果：

```
#analyzer-use-cython.py
import analyzer_cython as analyzer
import visualizer


data_dict = analyzer.load_json_data(analyzer.SOURCE_FILE)

results = analyzer.analyze_data(data_dict)

print "=== ( Shortest Answer ) === "
visualizer.displayShortestAnswer(results['shortest_answer'])

print "=== ( Most Active Users ) === "
```

```
visualizer.displayMostActiveUsers(results['most_active_users'])

print "=== ( Most Active Topics ) === "
visualizer.displayMostActiveTopics(results['most_active_topics'])

print "=== ( Most Helpful Users ) === "
visualizer.displayMostHelpfulUser(results['most_helpful_user'])

print "=== ( Most Answered Questions ) === "
visualizer.displayMostAnsweredQuestions(results['most_answered_questions'])

print "=== ( Most Common Phrases ) === "
visualizer.displayMostCommonPhrases(results['most_common_phrases'])
```

前面的代码可以通过下面的命令进行编译：

```
$ python analyzer-setup.py build_ext -inplace
```

然后我们运行analyzer-use-cython.py脚本，会看到下面的运行时间：

```
real    0m1.273s
user    0m1.203s
sys     0m0.068s
```

运行时间从3.5秒降到了1.3秒。这是通过简单地重组代码，使用Cython编译得到的显著的性能提升，就像我们在第6章看到的一样。经过简单的编译就可以获得很好的效果。

这段代码可以进一步分解，并对大部分使用复杂结构的代码进行改写，这样我们就可以把所有的变量都改成C语言原生类型。我们还可以把`nltk`换成C语言写的自然语言处理库，比如OpenNLP（http://opennlp.sourceforge.net/projects.html）。

8.3　小结

你已经看到了本章的结尾，也是本书结束的时候了。本章提供的示例，演示了如何通过前面几章学到的优化技术，对任意一段代码进行性能分析和改进。

就像并非所有的技术都可以彼此兼容一样，也不是所有的优化技术都能应用在本章的示例中。但是，我们看到了一些技术的使用，更具体地说，我们看到了多线程技术，`cProfile`和`kcachegrind`性能分析技术，以及通过Cython进行编译的技术。

感谢你花时间阅读本书，衷心地希望你喜欢它！

关注图灵教育 关注图灵社区

iTuring.cn

在线出版 电子书《码农》杂志 图灵访谈 ……



QQ联系我们

图灵读者官方群I：218139230
图灵读者官方群II：164939616



微博联系我们

官方账号：@图灵教育 @图灵社区 @图灵新知
市场合作：@图灵袁野
写作本版书：@图灵小花 @图灵张霞 @毛倩倩-图灵
翻译英文书：@朱巍ituring @楼伟珊
翻译日文书或文章：@图灵日语编辑部
翻译韩文书：@图灵陈曦
电子书合作：@hi_jeanne
图灵访谈/《码农》杂志：@刘敏ituring
加入我们：@王子是好人



微信联系我们



图灵教育
turingbooks



图灵访谈
ituring_interview

对于Python程序员来说，仅仅知道如何写代码是不够的，还要能够充分利用关键代码的处理能力。本书将讨论如何对Python代码进行性能分析，找出性能瓶颈，并通过不同的性能优化技术消除瓶颈。

本书从基本的概念开始，循序渐进地介绍高级的优化主题。首先介绍了Python的主流性能分析器，以及用于帮助理解性能分析结果的可视化工具。然后介绍了通用的性能优化方法和专门针对Python的性能优化方法，带你浏览该语言的主要结构，让你只需做一点改变，即可迅速改善代码的性能。最后介绍了一些专门用于数据处理的程序库，教你如何正确地使用它们以获得最佳性能。

如果你是一名Python开发者，想优化Python代码的性能，或是想进一步提升编程能力，那么本书非常适合你阅读。

通过阅读本书，你将能够：

- 掌握逐步优化代码的方法，学会使用不同的性能分析工具
- 理解性能分析器的概念，学会如何观察输出结果
- 利用性能分析工具解释可视化的性能输出结果，改善脚本的性能
- 用Cython快速创建Python与C语言混合的应用程序
- 利用PyPy改善Python代码的性能
- 通过Numba、Parakeet和pandas优化数据处理代码

Amazon读者推荐

“我是一名Python开发者，也是Python的骨灰粉。我觉得这本书中的内容非常实用，尤其是有关代码性能分析方法与工具的详细介绍。如果你想写出优雅、漂亮、高性能的Python代码，绝对应该看看这本书！”

“对于想进一步提升编程能力的Python程序员来说，这是一本非常值得购买的书。‘高性能Python’是一个内容十分丰富的主题，而这本书深入探讨了Python代码的性能关键点。”

[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/软件开发/Python
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-42422-8
定价：45.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks